Advanced Data Structures and Algorithm Analysis

丁尧相
浙江大学

Spring & Summer 2024
Lecture 12
2024-5-13

Outline: Local Search

- Gradient descent
- Metropolis algorithm and simulated annealing
- Hopfield neural networks
- Max-cut problem
- Take-home messages

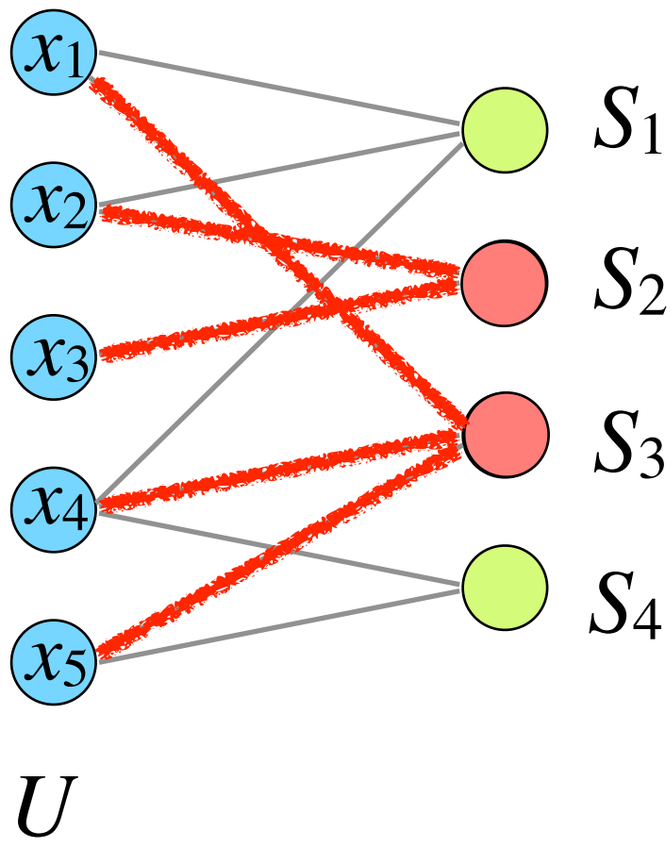
Outline: Local Search

- Gradient descent
- Metropolis algorithm and simulated annealing
- Hopfield neural networks
- Max-cut problem
- Take-home messages

Set Cover

Instance: A number of sets $S_1, S_2, ..., S_m \subseteq U$.

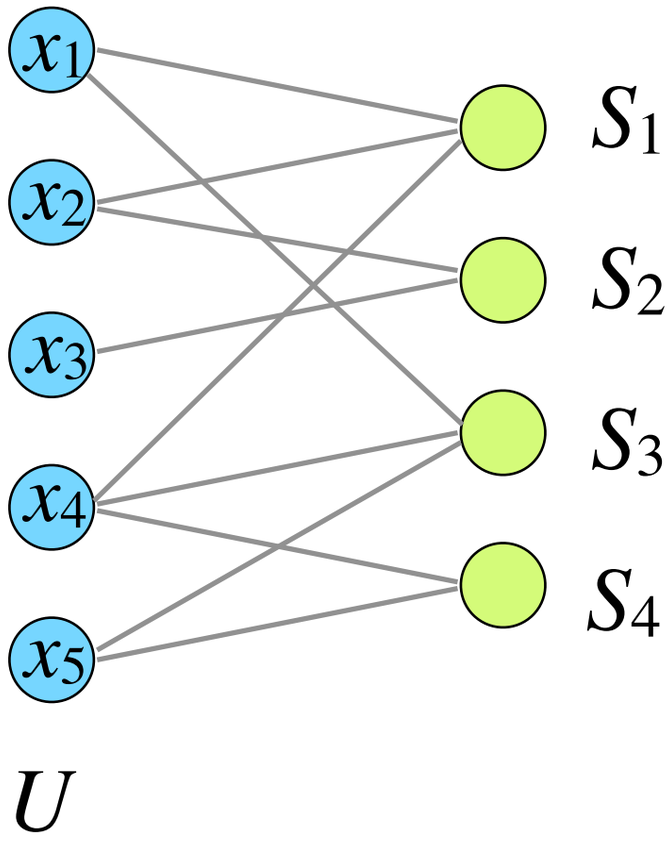
Find the smallest $C \subseteq \{1, 2, ..., m\}$ that $\cup_{i \in C} S_i = U$.



Set Cover

Instance: A number of sets $S_1, S_2, ..., S_m \subseteq U$.

Find the smallest $C \subseteq \{1, 2, ..., m\}$ that $\cup_{i \in C} S_i = U$.



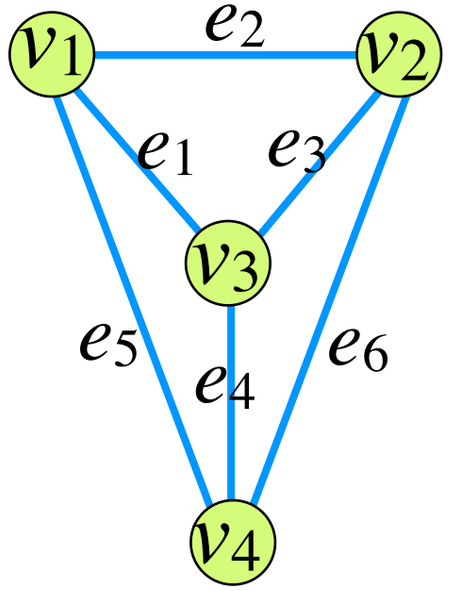
- **NP**-hard
- one of Karp's 21 **NP**-complete problems
- *frequency*: # of sets an element is in

$$frequency(x) = |\{S_i : x \in S_i\}|$$

Vertex Cover

Instance: An undirected graph $G(V,E)$

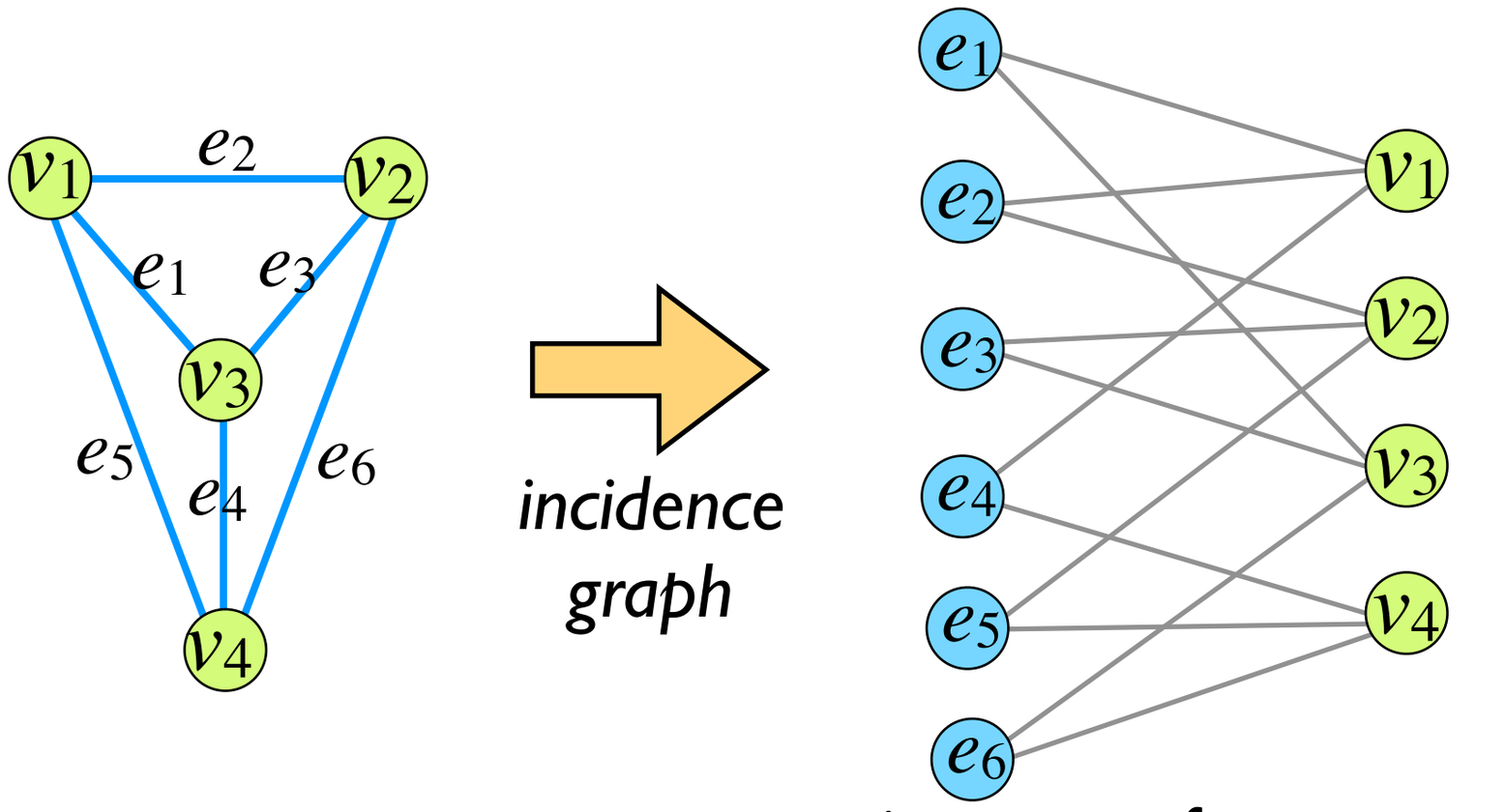
Find the smallest $C \subseteq V$ that every edge has at least one endpoint in C.



Vertex Cover

Instance: An undirected graph $G(V,E)$

Find the smallest $C \subseteq V$ that every edge has at least one endpoint in C.



Vertex Cover

Instance: An undirected graph $G(V,E)$

Find the smallest $C \subseteq V$ that every edge has at least one endpoint in C.

- **NP**-hard
- one of Karp's 21 **NP**-complete problems

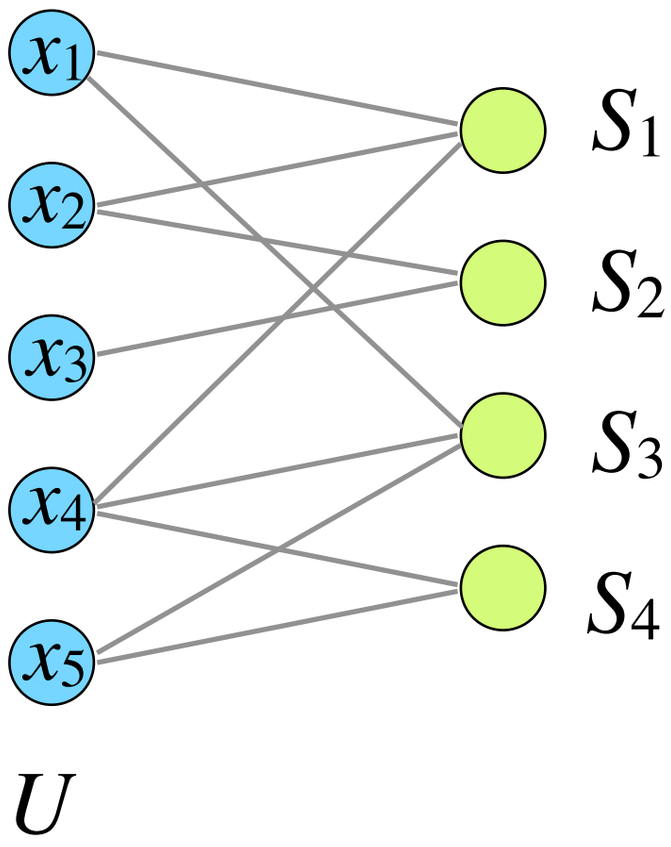
VC is **NP**-hard $\Rightarrow$ SC is **NP**-hard

Proved in Lecture 10.

Set Cover

Instance: A number of sets $S_1, S_2, ..., S_m \subseteq U$.

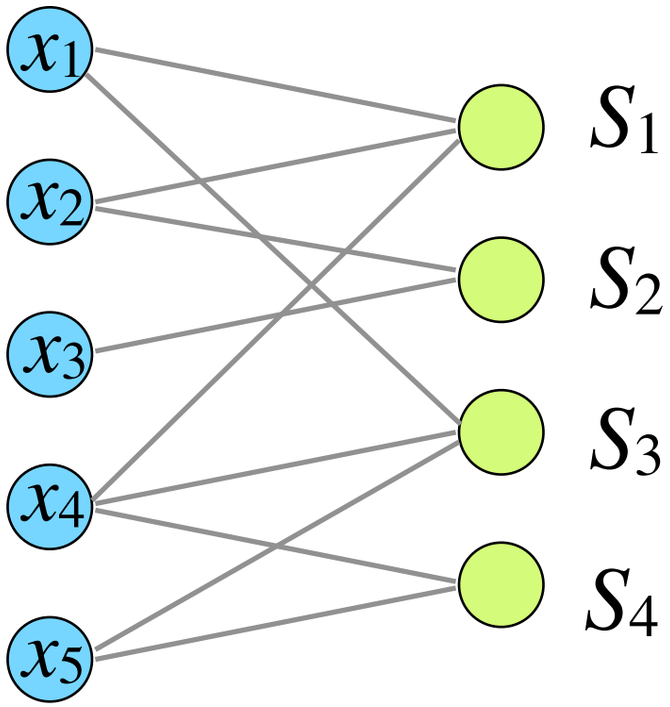
Find the smallest $C \subseteq \{1, 2, ..., m\}$ that $\cup_{i \in C} S_i = U$.



GreedyCover

Initially $C=\emptyset$;
while $U \neq \emptyset$ do:
  add i with largest $|S_i \cap U|$ to C;
  $U = U \setminus S_i$;

Instance: A number of sets $S_1, S_2, \ldots, S_m \subseteq U$.



GreedyCover

Initially $C=\emptyset$;
while $U \neq \emptyset$ do:
  add i with largest $|S_i \cap U|$ to C;
  $U = U \setminus S_i$;

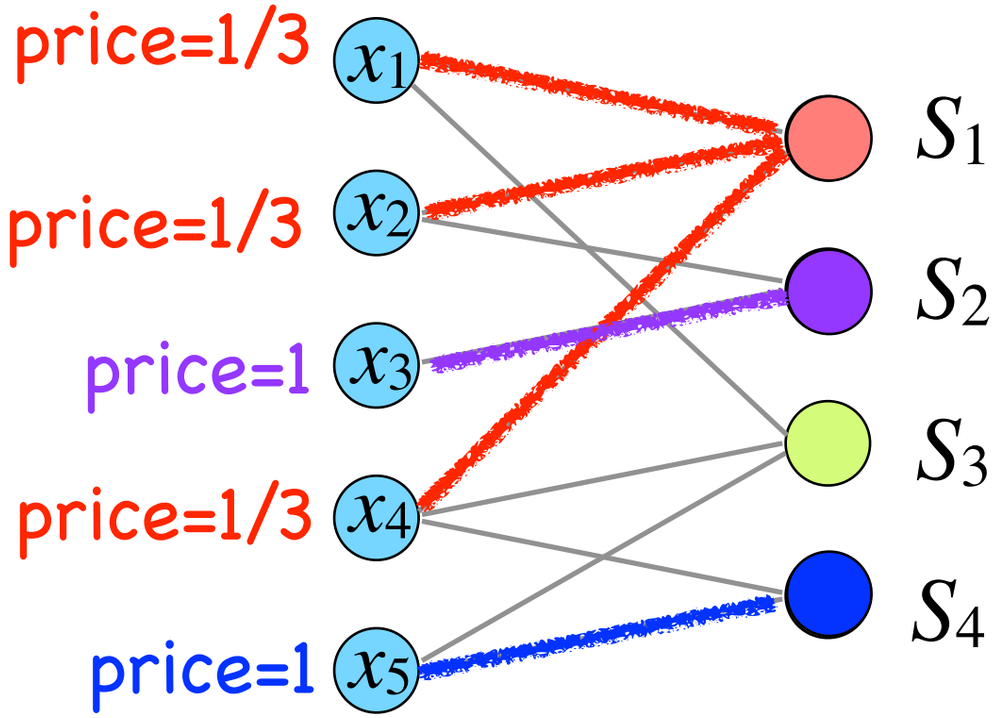
OPT(I): value of minimum set cover of instance I

SOL(I): value of the set cover returned by the **GreedyCover** algorithm on instance I

GreedyCover has *approximation ratio* α if

$$\forall \text{ instance } I, \quad \frac{\mathrm{SOL}(I)}{\mathrm{OPT}(I)} \leq \alpha$$

Instance: A number of sets $S_1, S_2, \ldots, S_m \subseteq U$.



GreedyCover

Initially $C=\emptyset$;
while $U \neq \emptyset$ do:
  add i with largest $|S_i \cap U|$ to C;
  $U = U \setminus S_i$; ⌜∀x∈Sᵢ∩U, price(x)=1/|Sᵢ∩U|

$$|C| = \sum_{x \in U} price(x)$$

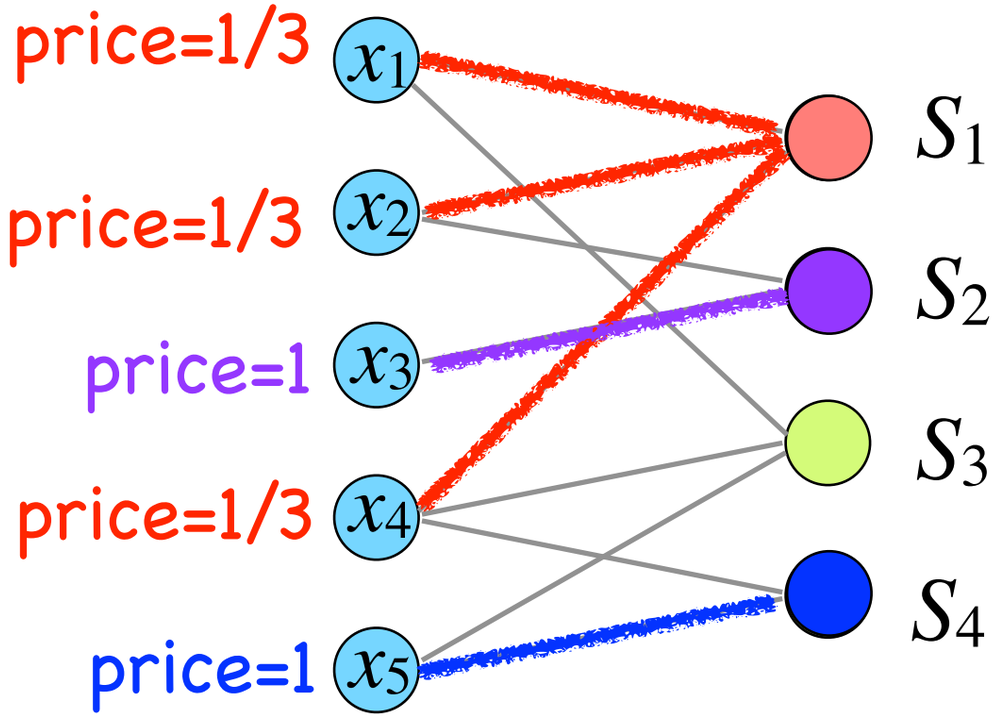
enumerate $x_1, x_2, \ldots x_n$ in the order in which they are covered

elements can be *matched* to the sets in OPT cover

$$\exists S_i, \ |S_i| \geq \frac{|U|}{OPT} \Rightarrow price(x_1) \leq \frac{OPT}{|U|}$$

See https://tcs.nju.edu.cn/wiki/index.php?title=%E9%AB%98%E7%BA%A7%E7%AE%97%E6%B3%95_(Fall_2020)/Greedy_and_Local_Search

Instance: A number of sets $S_1, S_2, \ldots, S_m \subseteq U$.



GreedyCover

Initially $C=\emptyset$;
while $U\neq\emptyset$ do:
  add i with largest $|S_i \cap U|$ to C;
  $U = U \setminus S_i$; ∀x∈S_i, price(x)=1/$|S_i \cap U|$

$$|C| = \sum_{x \in U} price(x)$$

enumerate $x_1, x_2, \ldots x_n$ in the order in which they are covered

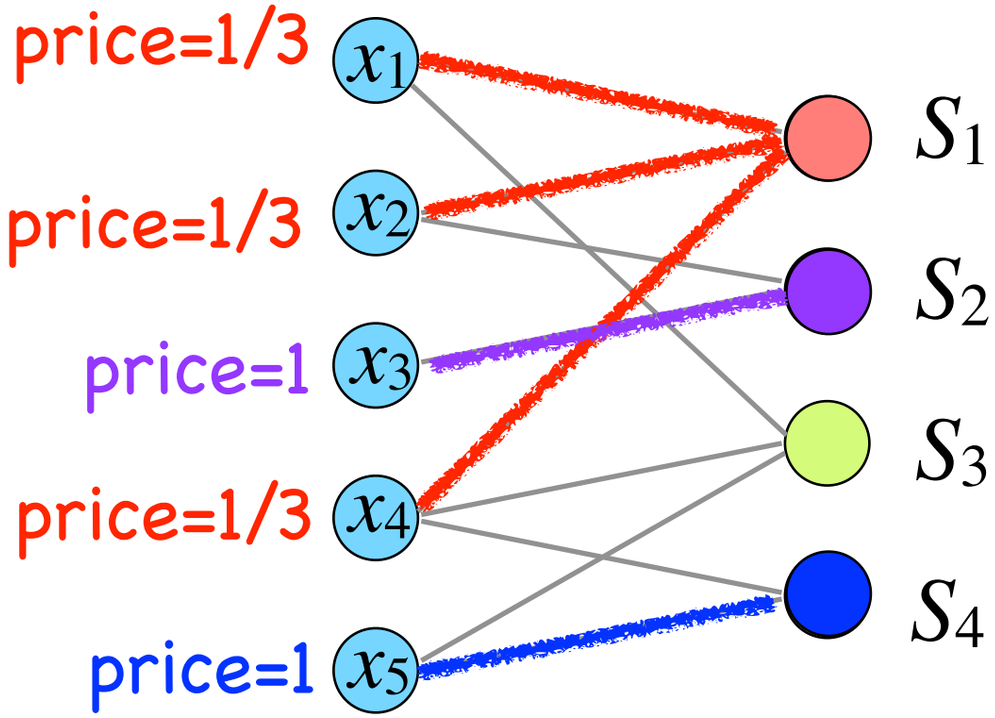
$$price(x_k) \leq \frac{OPT}{n-k+1}$$

consider U_t in iteration t where x_k is covered:

$|U_t| \geq n\text{-}k\text{+}1$

all $S_i \cap U_t$ form a set cover instance: ≤OPT

Instance: A number of sets $S_1, S_2, \ldots, S_m \subseteq U$.



GreedyCover

Initially $C=\emptyset$;
while $U \neq \emptyset$ do:
  add i with largest $|S_i \cap U|$ to C;
  $U = U \setminus S_i$; $\forall x \in S_i$, $price(x) = 1/|S_i \cap U|$

$$|C| = \sum_{x \in U} price(x) \leq \sum_{k=1}^{n} \frac{OPT}{n-k+1} = H_n \cdot OPT$$

enumerate $x_1, x_2, \ldots x_n$ in the order in which they are covered

$$price(x_k) \leq \frac{OPT}{n-k+1}$$

GreedyCover

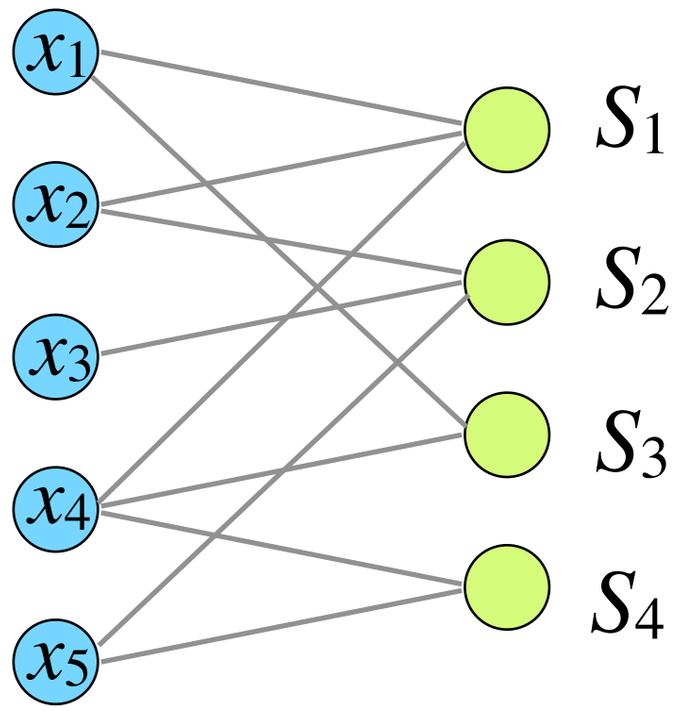
Initially $C=\emptyset$;
while $U\neq\emptyset$ do:
  add i with largest $|S_i\cap U|$ to C;
  $U = U \setminus S_i$;

- *GreedyCover* has approximation ratio $H_n \approx \ln n + O(1)$.
- [Lund, Yannakakis 1994; Feige 1998] There is no poly-time (1-o(1))ln n-approx. algorithm unless **NP** = quasi-poly-time.
- [Ras, Safra 1997] For some c there is no poly-time c ln n-approximation algorithm unless **NP = P**.
- [Dinur, Steuer 2014] There is no poly-time (1-o(1))ln n-approximation algorithm unless **NP = P**.

Instance: A number of sets $S_1, S_2, \ldots, S_m \subseteq U$.

Primal: $C \subseteq \{1, 2, \ldots, m\}$ that $\cup_{i \in C} S_i = U$.

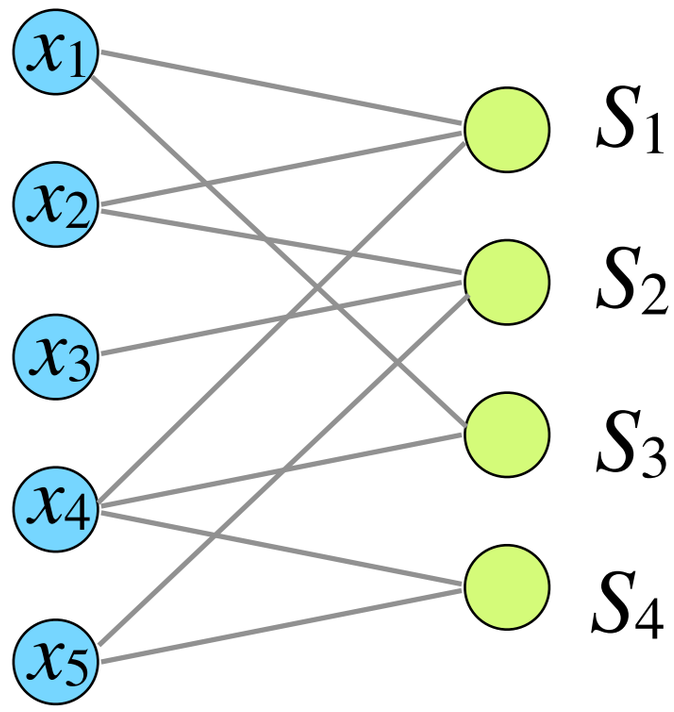
$\text{OPT}_{\text{primal}} = \min |\, C \,|$



Instance: A number of sets $S_1, S_2, ..., S_m \subseteq U$.

Primal: $C \subseteq \{1, 2, ..., m\}$ that $\cup_{i \in C} S_i = U$.

$$\text{OPT}_{\text{primal}} = \min |\, C \,|$$



Dual: $M \subseteq U$ that $\forall i, |S_i \cap M| \leq 1$.

$\forall C, \forall M$: $|\, M \,| \leq |\, C \,|$

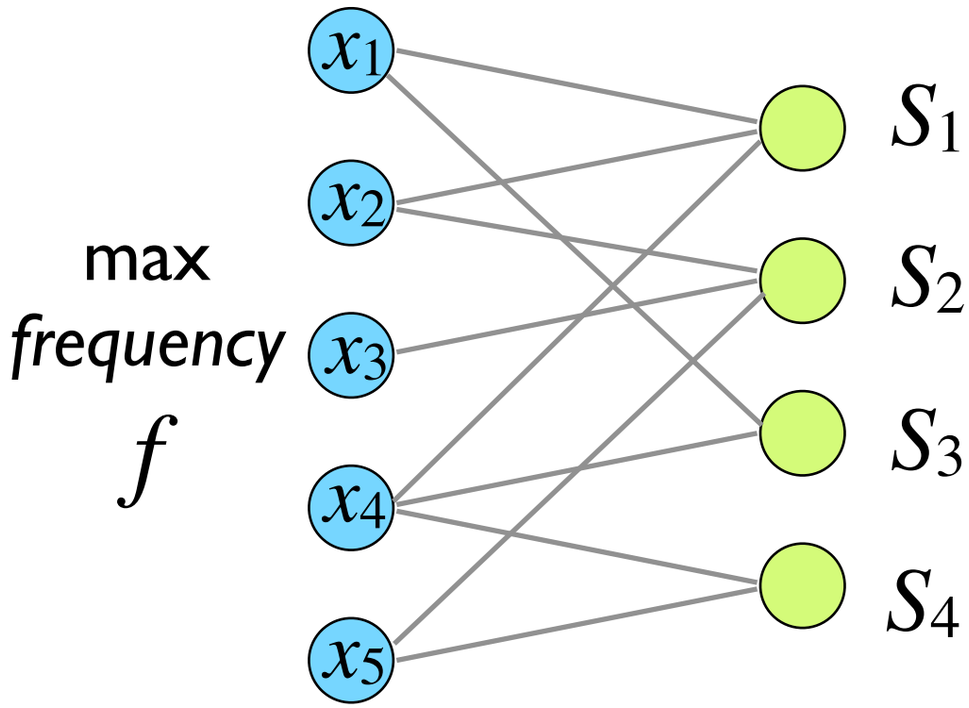
every $x \in M$ must consume a set to cover

$$\forall M: |\, M \,| \leq \text{OPT}_{\text{primal}}$$

Instance: A number of sets $S_1, S_2, ..., S_m \subseteq U$.

Primal: $C \subseteq \{1, 2, ..., m\}$ that $\cup_{i \in C} S_i = U$.

$$\text{OPT}_{\text{primal}} = \min |\, C \,|$$



Find a *maximal* M;
return $C = \{i : S_i \cap M \neq \emptyset\}$;

M is *maximal* $\Rightarrow$ C must be a cover

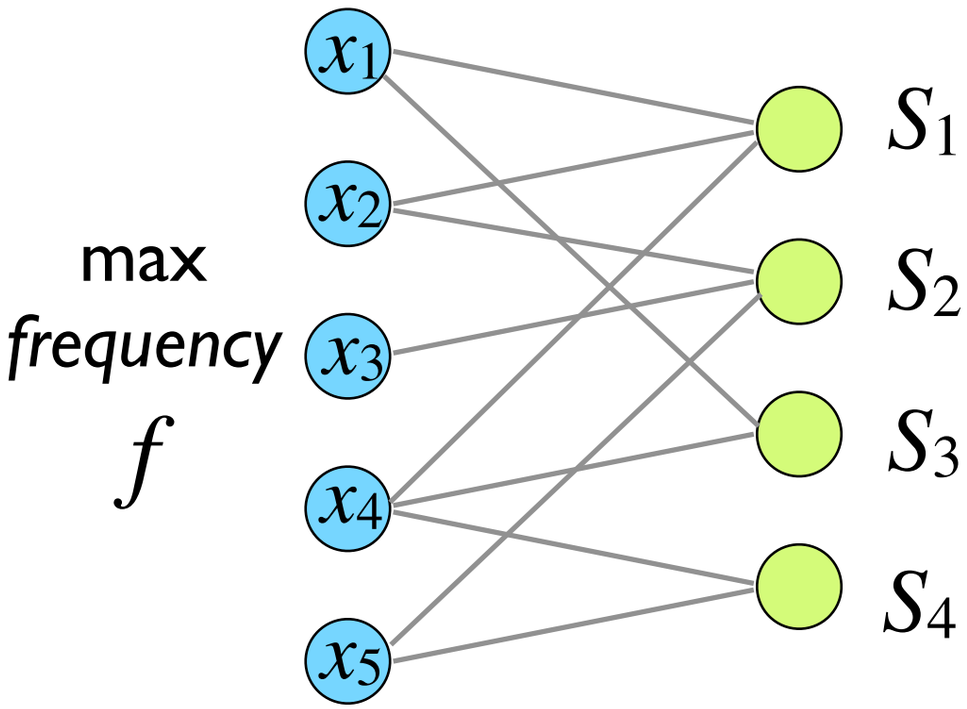
$$|C| \leq f \cdot |M| \leq f \cdot \text{OPT}_{\text{primal}}$$

Dual: $M \subseteq U$ that $\forall i, |S_i \cap M| \leq 1$.

$$\forall M: \; |\, M \,| \leq \text{OPT}_{\text{primal}}$$

Instance: A number of sets $S_1, S_2, \ldots, S_m \subseteq U$.

Find a *maximal* $M \subseteq U$ that $\forall i, |S_i \cap M| \leq 1$;
return $C = \{i : S_i \cap M \neq \emptyset\}$;



Frequency assumption:

$$\forall x \in U,\ |\{i: x \in S_i\}| \leq f$$

$$|C| \leq f \cdot \mathrm{OPT}$$

For vertex cover: This gives a 2-approximation algorithm.

Vertex Cover

Instance: An undirected graph $G(V,E)$

Find the smallest $C \subseteq V$ that every edge has at least one endpoint in C.

a 2-approximation algorithm:

Find a *maximal matching*;
return the *matched* vertices;

- [Dinur, Safra 2005] There is no poly-time <1.36-approximation algorithm unless **NP = P**.
- [Khot, Regev 2008] Assuming the unique games conjecture, there is no poly-time (2-ε)-approximation algorithm.

Solve problems *approximately*

Solve problems *approximately*

—— aims at a local optimum

Solve problems *approximately*

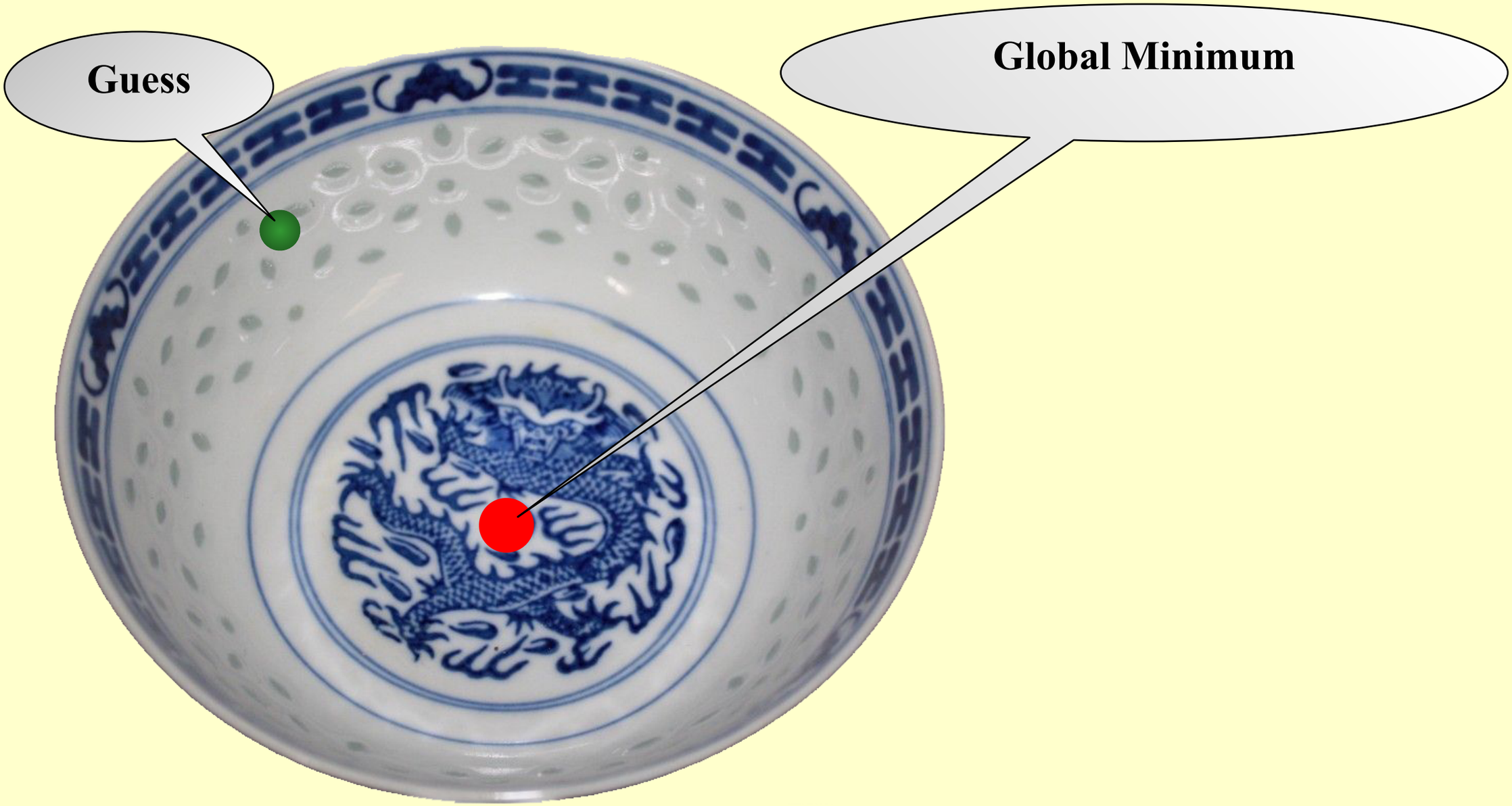
—— aims at a local optimum

Solve problems *approximately*

—— aims at a local optimum

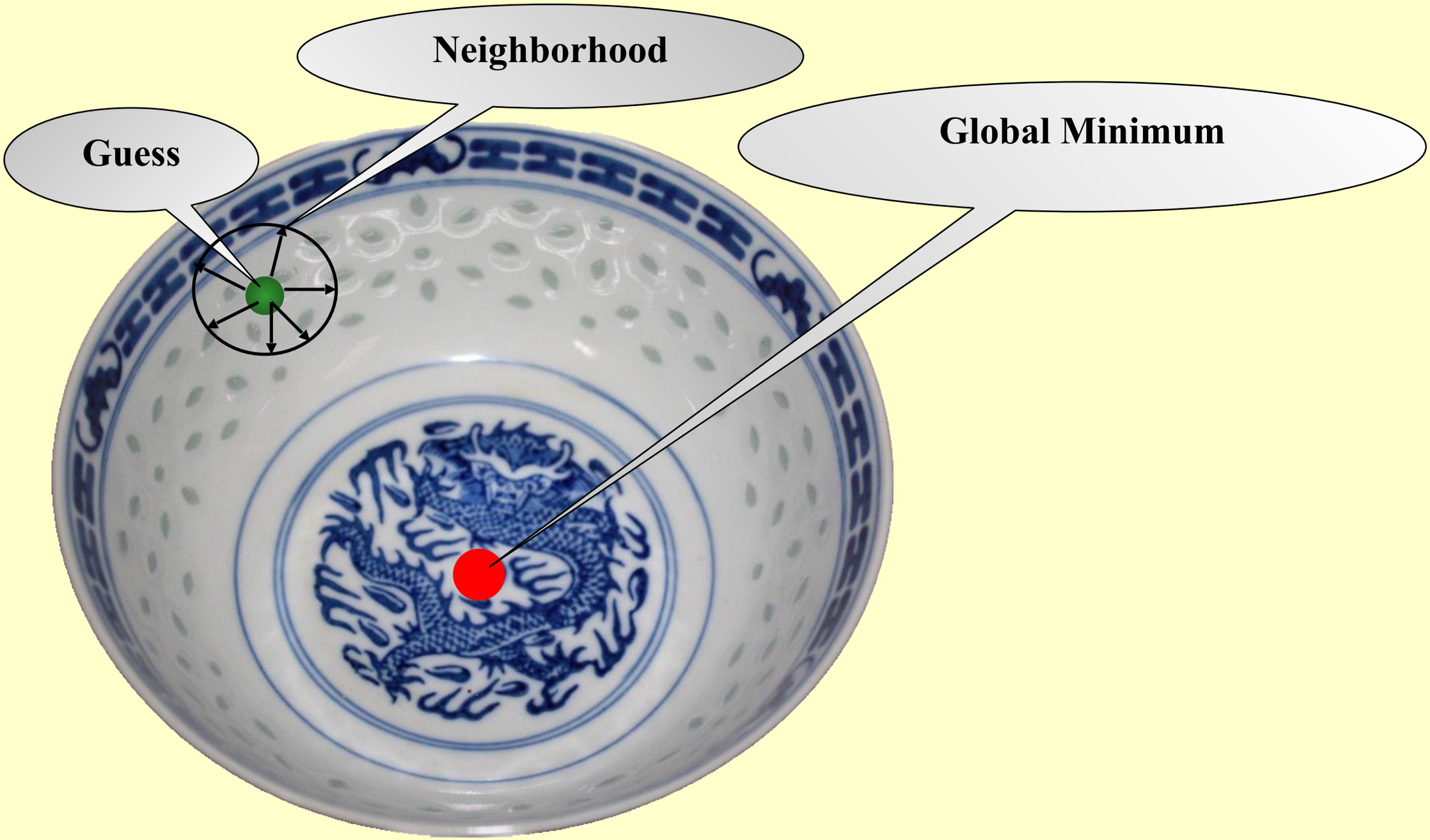
Solve problems *approximately*

—— aims at a local optimum



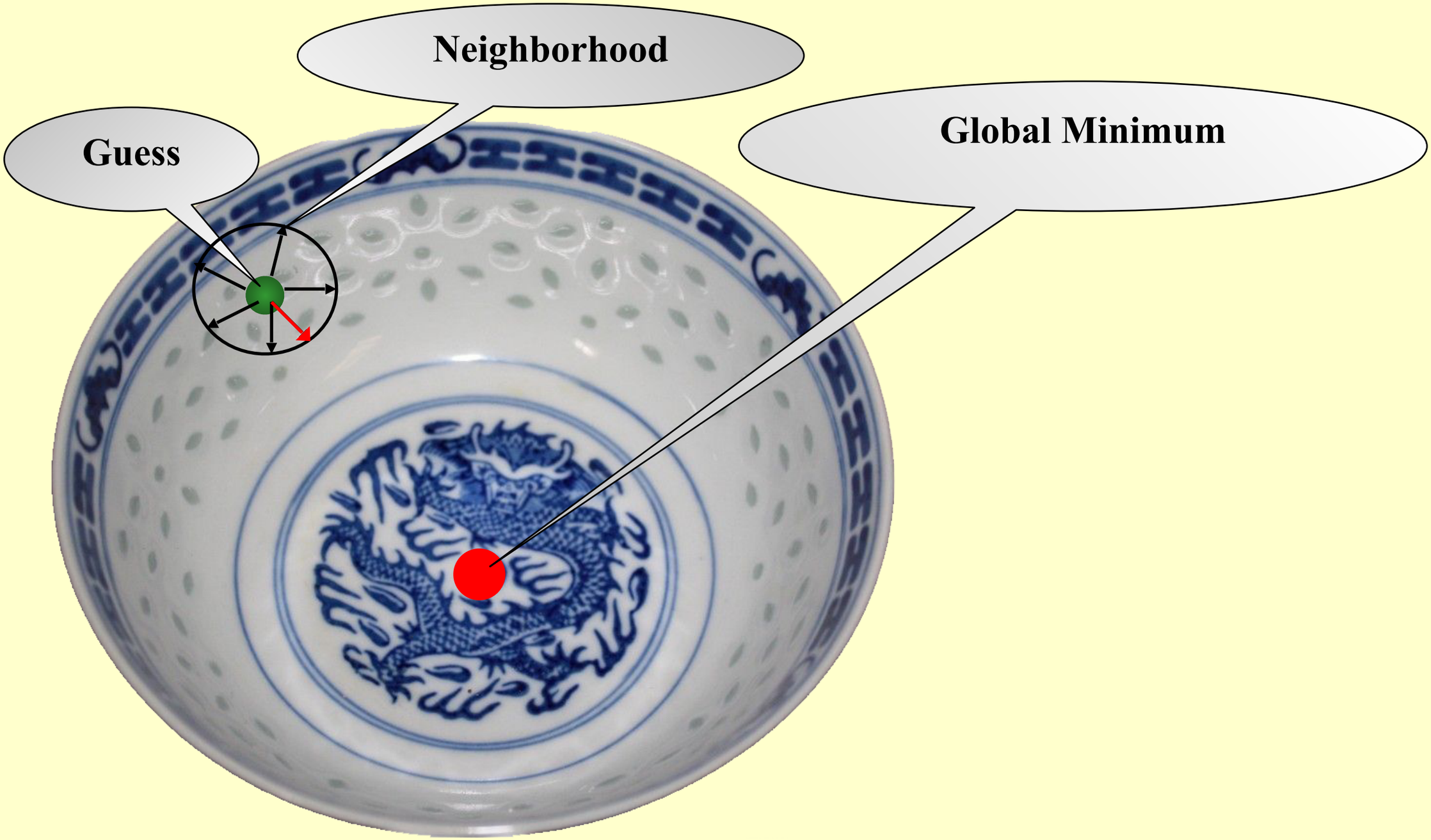
Solve problems *approximately*

—— aims at a local optimum



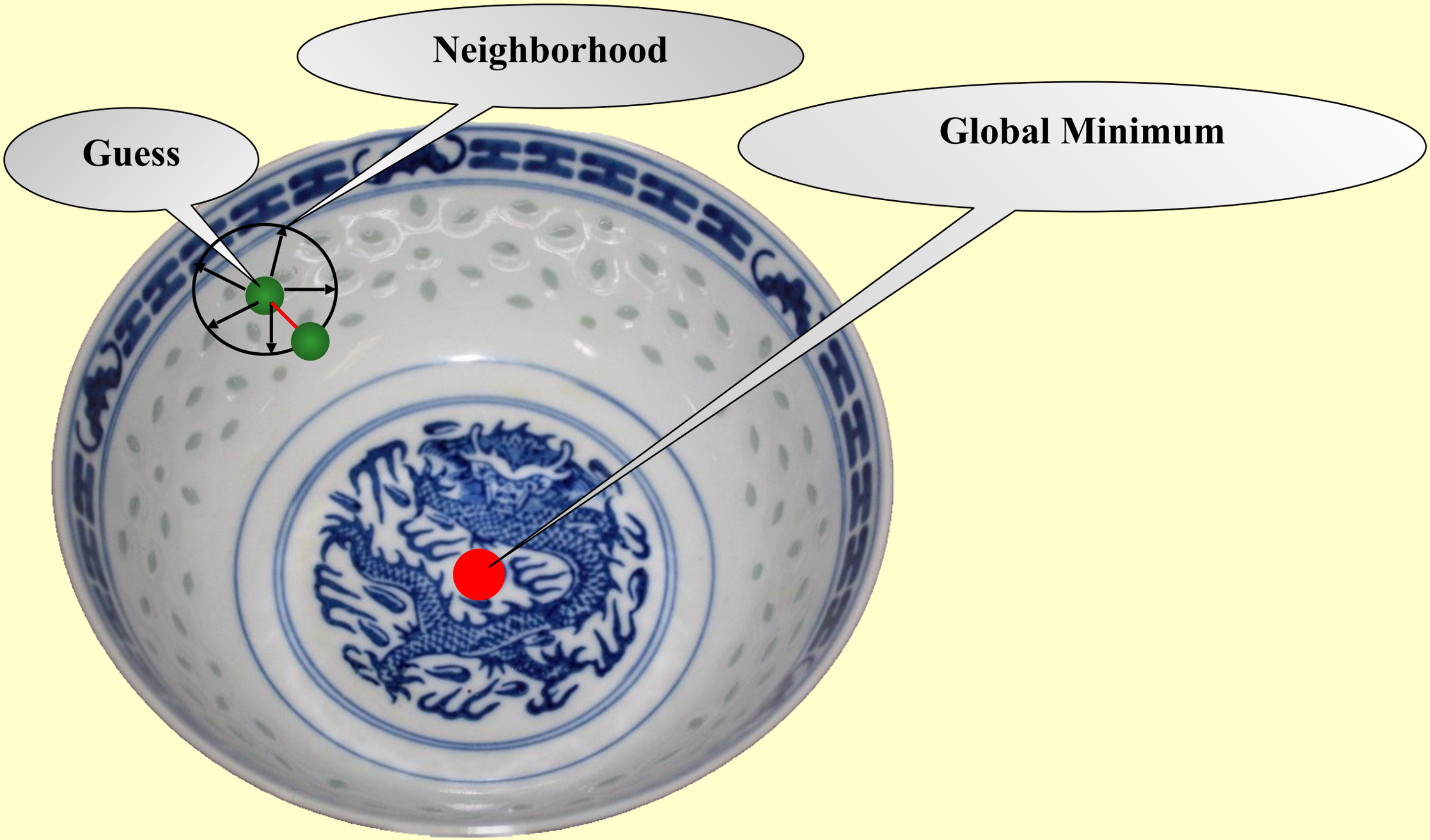
Solve problems *approximately*

—— aims at a local optimum



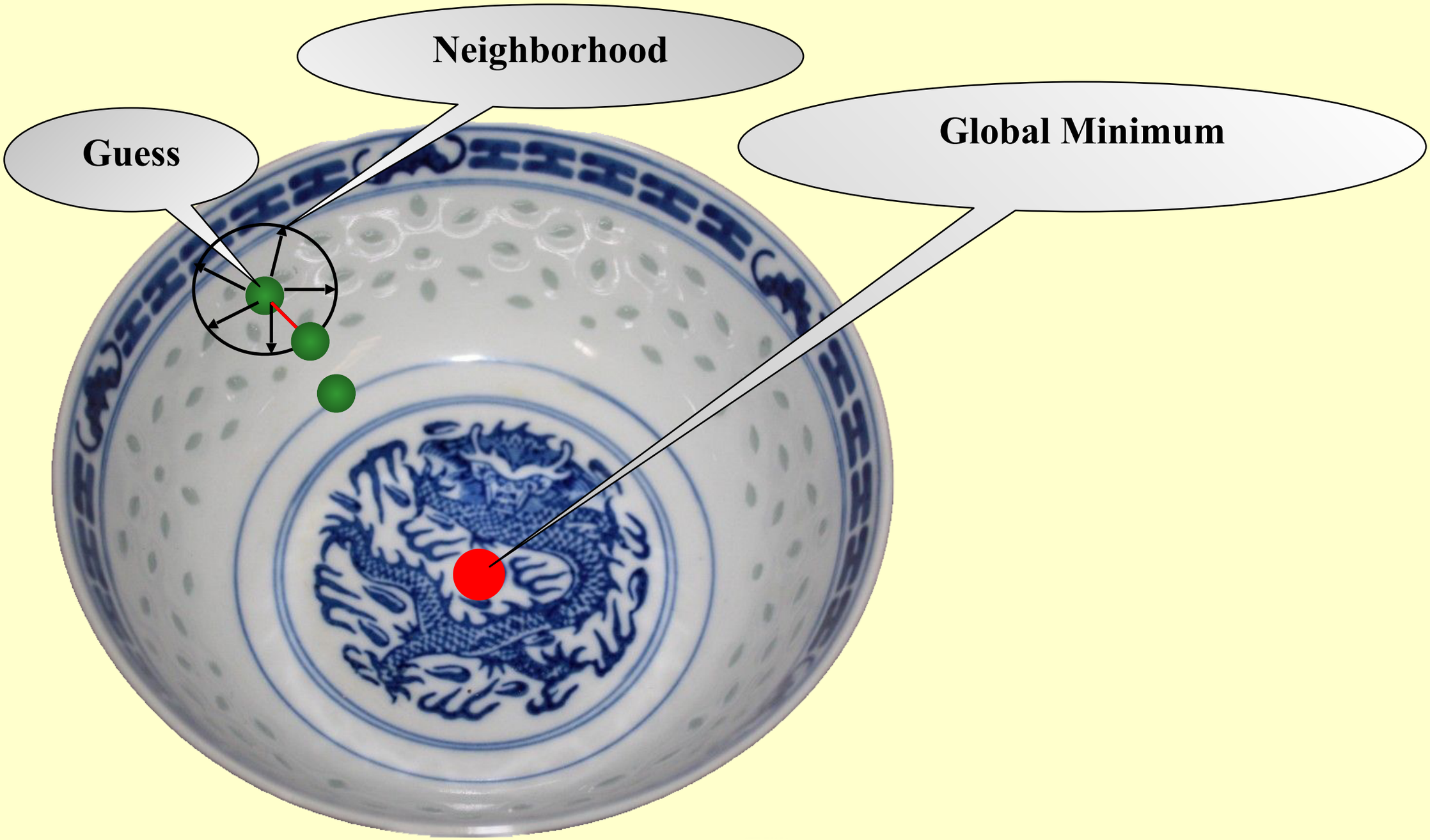
Solve problems *approximately*

—— aims at a local optimum



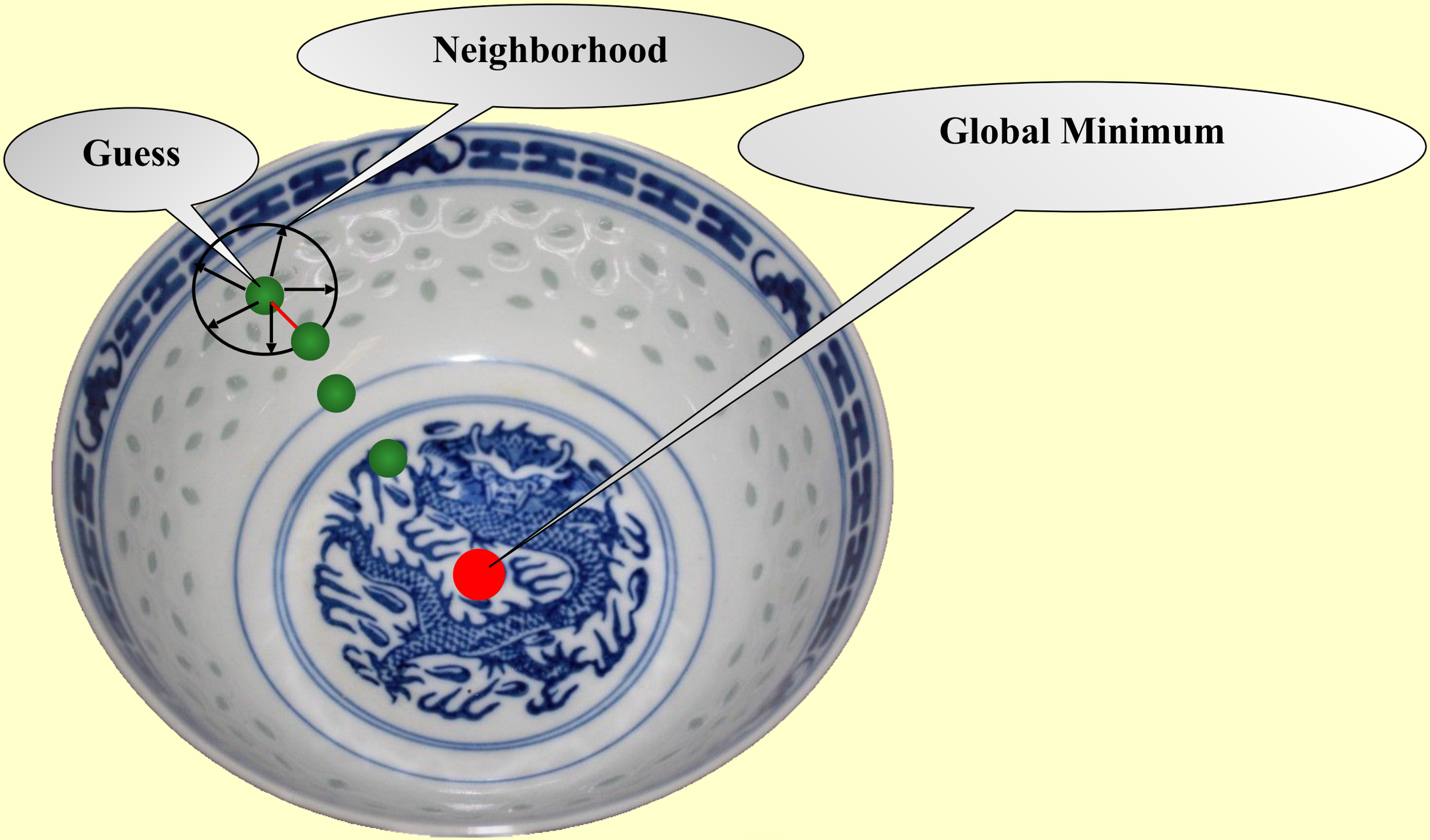
Solve problems *approximately*

—— aims at a local optimum



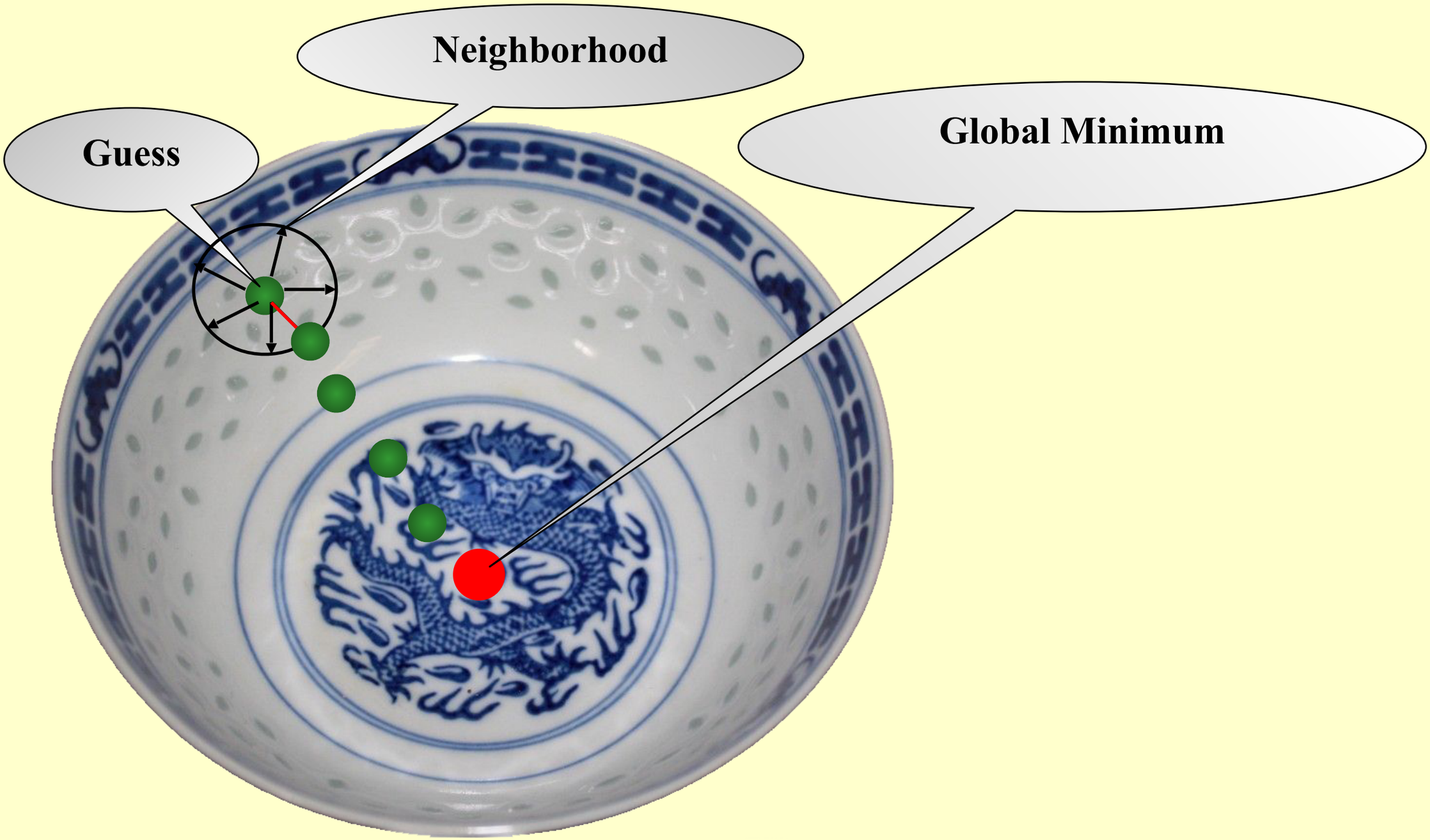
Solve problems *approximately*

—— aims at a local optimum



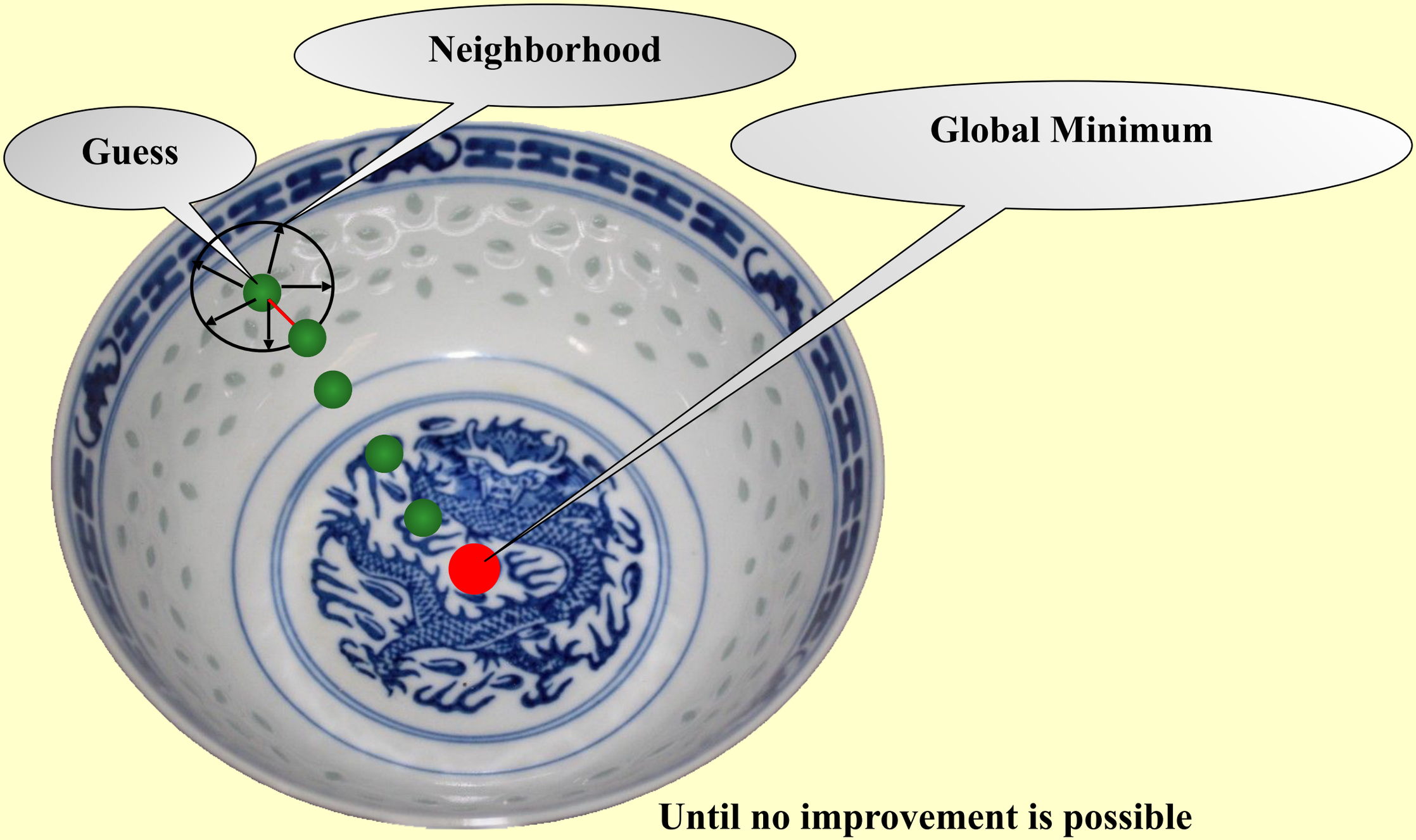
Solve problems *approximately*

—— aims at a local optimum



Solve problems *approximately*

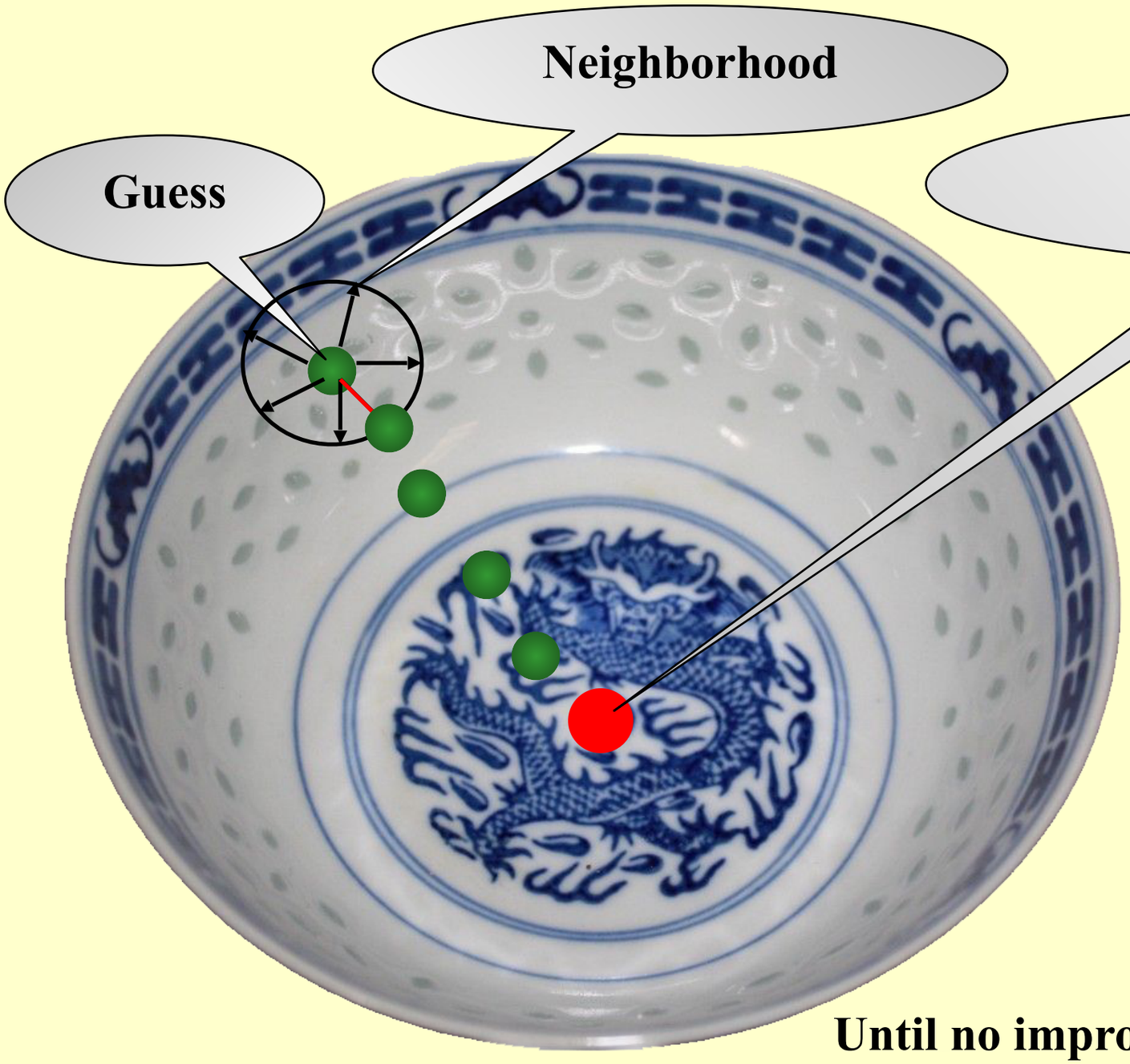
—— aims at a local optimum

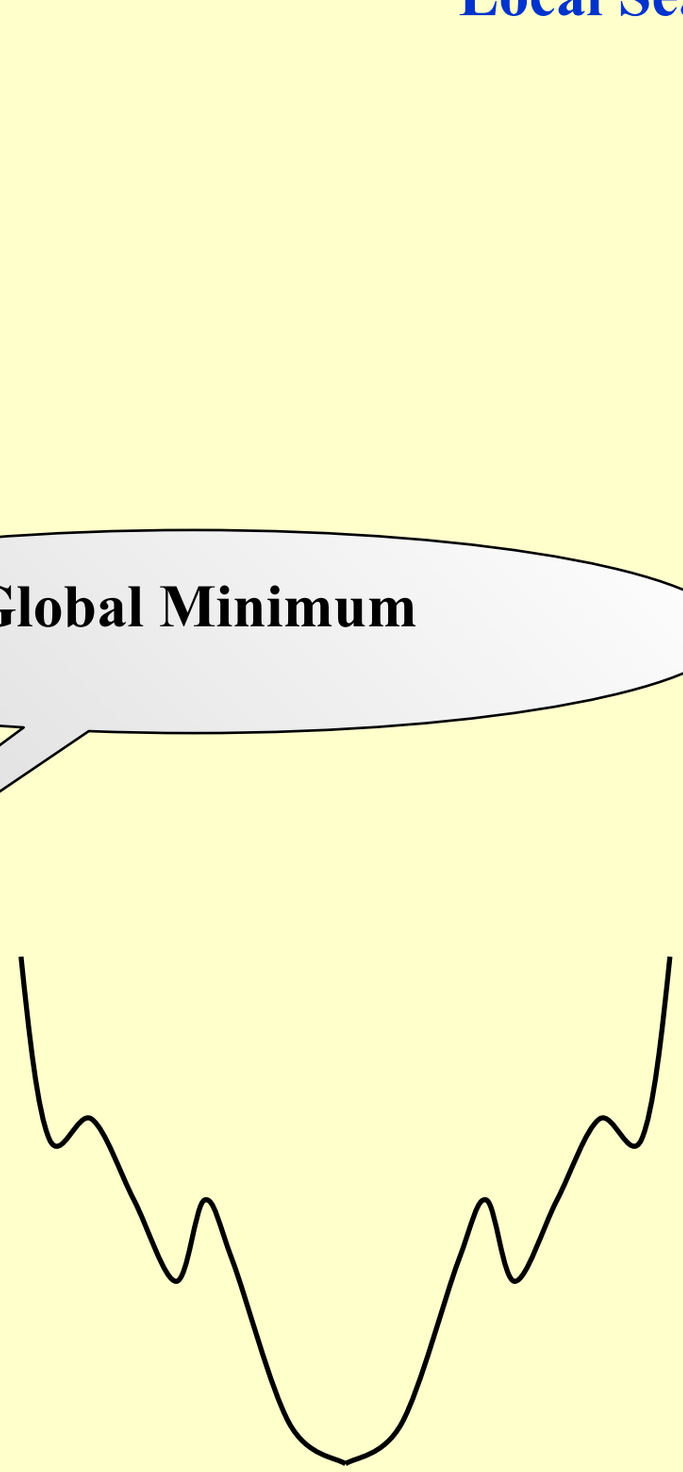


Until no improvement is possible

Solve problems *approximately*

—— aims at a local optimum

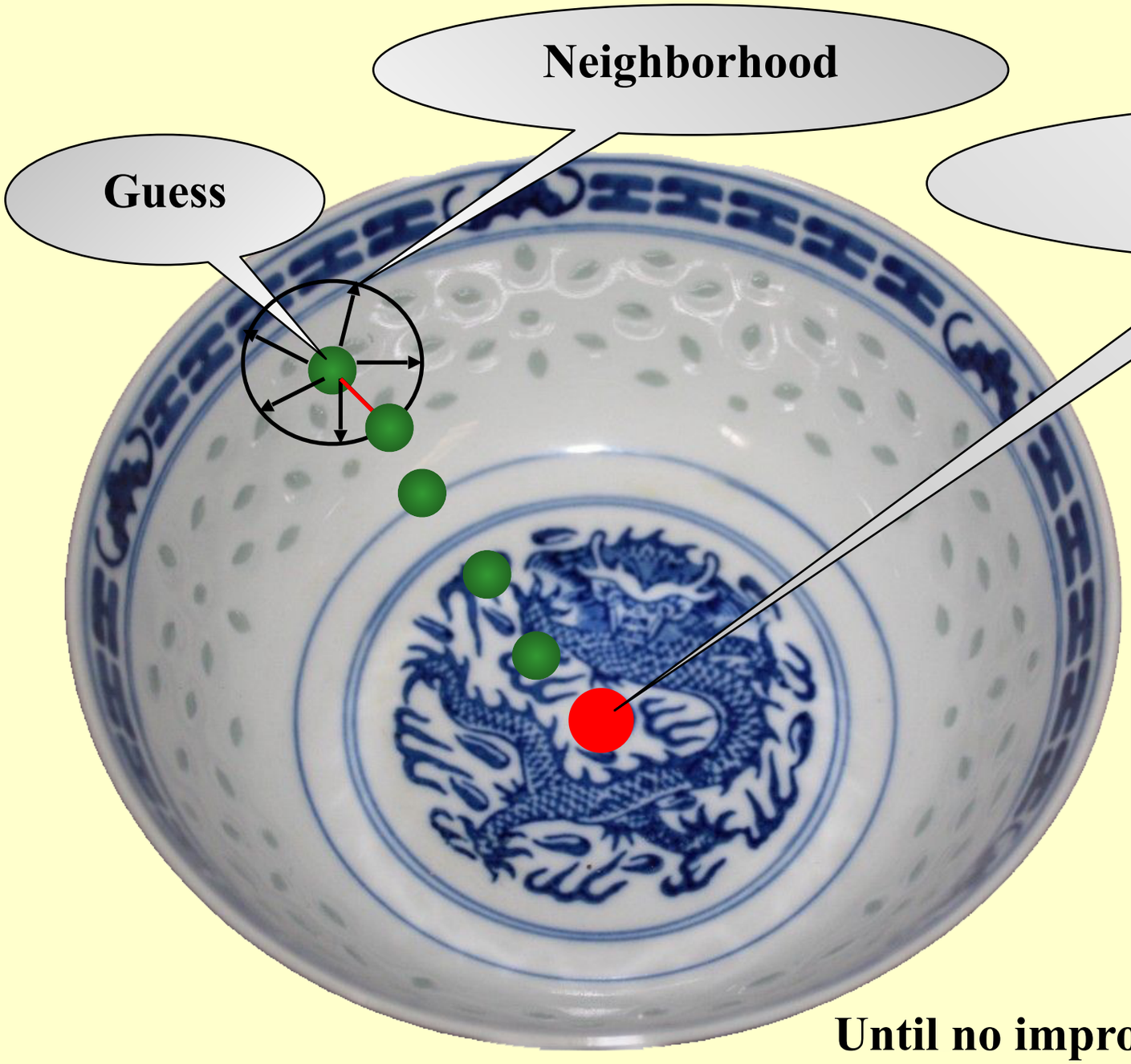


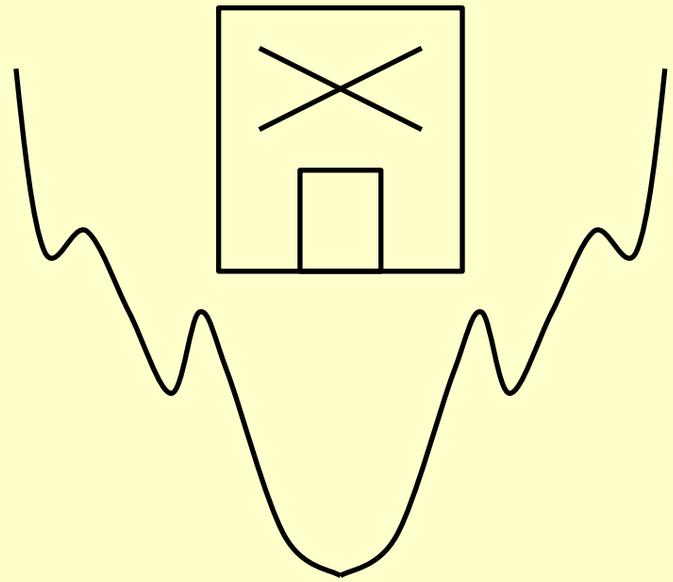


Until no improvement is possible

Solve problems *approximately*

—— aims at a local optimum

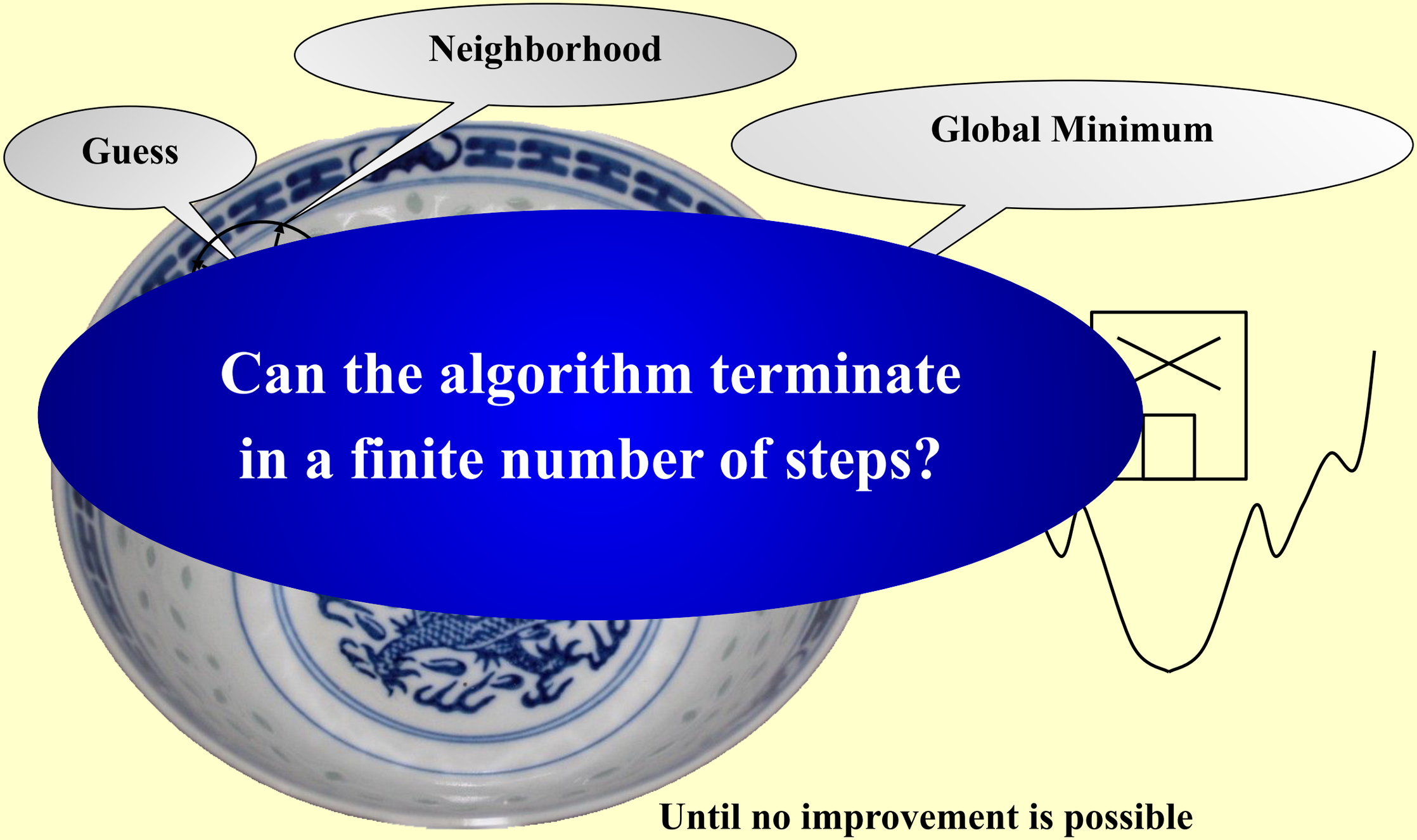




Until no improvement is possible

Solve problems *approximately*

—— aims at a local optimum



Until no improvement is possible

Framework of Local Search

Framework of Local Search

☞ **Local**

Framework of Local Search

☞ **Local**

☞ **Search**

Framework of Local Search

☞ **Local**

- **Define *neighborhoods* in the feasible set**
- **A local optimum is a best solution in a neighborhood**

☞ **Search**

Framework of Local Search

☞ **Local**

- **Define *neighborhoods* in the feasible set**
- **A local optimum is a best solution in a neighborhood**

☞ **Search**

- **Start with a feasible solution and search a better one within the neighborhood**
- **A local optimum is achieved if no improvement is possible**

Neighbor Relation

Neighbor Relation

☞ **S ~ S' : S' is a *neighboring solution* of S – S' can be obtained by a small modification of S.**

Neighbor Relation

- ☞ **S ~ S' : S' is a *neighboring solution* of S – S' can be obtained by a small modification of S.**
- ☞ **N(S): *neighborhood* of S – the set { S': S ~ S' }.**

Neighbor Relation

☞ **S ~ S' : S' is a *neighboring solution* of S – S' can be obtained by a small modification of S.**

☞ **N(S): *neighborhood* of S – the set { S': S ~ S' }.**

```
SolutionType Gradient_descent()
{   Start from a feasible solution S ∈ FS ;
    MinCost = cost(S);
    while (1) {
        S' = Search( N(S) ); /* find the best S' in N(S) */
        CurrentCost = cost(S');
        if ( CurrentCost < MinCost ) {
            MinCost = CurrentCost;    S = S';
        }
        else  break;
    }
    return S;
}
```

Neighbor Relation

☞ S ~ S' : S' is a *neighboring solution* of S – S' can be obtained by a small modification of S.

☞ N(S): *neighborhood* of S – the set { S': S ~ S' }.

```
SolutionType Gradient_descent()
{   Start from a feasible solution S ∈ FS ;
    MinCost = cost(S);
    while (1) {
        S' = Search( N(S) ); /* find the best S' in N(S) */
        CurrentCost = cost(S');
        if ( CurrentCost < MinCost ) {
            MinCost = CurrentCost;    S = S';
        }
        else  break;
    }
    return S;
}
```

Neighbor Relation

☞ S ~ S' : S' is a *neighboring solution* of S – S' can be obtained by a small modification of S.

☞ N(S): *neighborhood* of S – the set { S': S ~ S' }.

```
SolutionType Gradient_descent()
{   Start from a feasible solution S ∈ FS ;
    MinCost = cost(S);
    while (1) {
        S' = Search( N(S) ); /* find the best S' in N(S) */
        CurrentCost = cost(S');
        if ( CurrentCost < MinCost ) {
            MinCost = CurrentCost;    S = S';
        }
        else  break;
    }
    return S;
}
```

〖Example〗 The Vertex Cover Problem.

〖Example〗 The Vertex Cover Problem.

❖ **Vertex cover problem:** Given an undirected graph G = (V, E) and an integer K, does G contain a subset V' ⊆ V such that |V'| is (at most) K and every edge in G has a vertex in V' (*vertex cover*)?

〖Example〗 The Vertex Cover Problem.

❖ Vertex cover problem: Given an undirected graph G = (V, E) and an integer K, does G contain a subset V' ⊆ V such that |V'| is (at most) K and every edge in G has a vertex in V' (*vertex cover*)?

❖ **Vertex cover problem: Given an undirected graph G = (V, E). Find a *minimum* subset S of V such that for each edge (*u*, *v*) in E, either *u* or *v* is in S.**

〖Example〗 The Vertex Cover Problem.

❖ Vertex cover problem: Given an undirected graph G = (V, E) and an integer K, does G contain a subset V' ⊆ V such that |V'| is (at most) K and every edge in G has a vertex in V' (*vertex cover*)?

❖ **Vertex cover problem: Given an undirected graph G = (V, E). Find a *minimum* subset S of V such that for each edge (*u*, *v*) in E, either *u* or *v* is in S.**

☞ **Feasible solution set $\mathcal{FS}$: all the vertex covers. $V \in \mathcal{FS}$**

〖Example〗 The Vertex Cover Problem.

❖ Vertex cover problem: Given an undirected graph G = (V, E) and an integer K, does G contain a subset V' ⊆ V such that |V'| is (at most) K and every edge in G has a vertex in V' (*vertex cover*)?

❖ **Vertex cover problem: Given an undirected graph G = (V, E). Find a *minimum* subset S of V such that for each edge (*u*, *v*) in E, either *u* or *v* is in S.**

☞ **Feasible solution set $\mathcal{FS}$: all the vertex covers. $V \in \mathcal{FS}$**

〖Example〗 The Vertex Cover Problem.

❖ Vertex cover problem: Given an undirected graph G = (V, E) and an integer K, does G contain a subset V' ⊆ V such that |V'| is (at most) K and every edge in G has a vertex in V' (*vertex cover*)?

❖ Vertex cover problem: Given an undirected graph G = (V, E). Find a *minimum* subset S of V such that for each edge (*u*, *v*) in E, either *u* or *v* is in S.

☞ Feasible solution set $\mathcal{FS}$: all the vertex covers. $V \in \mathcal{FS}$

☞ cost(S) = | S |

〖Example〗 The Vertex Cover Problem.

❖ Vertex cover problem: Given an undirected graph G = (V, E) and an integer K, does G contain a subset V' ⊆ V such that |V'| is (at most) K and every edge in G has a vertex in V' (*vertex cover*)?

❖ Vertex cover problem: Given an undirected graph G = (V, E). Find a *minimum* subset S of V such that for each edge (*u*, *v*) in E, either *u* or *v* is in S.

☞ Feasible solution set $\mathcal{FS}$: all the vertex covers. $V \in \mathcal{FS}$

☞ cost(S) = | S |

☞ S ~ S': S'

dele

〖Example〗 The Vertex Cover Problem.

❖ Vertex cover problem: Given an undirected graph G = (V, E) and an integer K, does G contain a subset V' ⊆ V such that |V'| is (at most) K and every edge in G has a vertex in V' (*vertex cover*)?

❖ **Vertex cover problem: Given an undirected graph G = (V, E). Find a *minimum* subset S of V such that for each edge (*u*, *v*) in E, either *u* or *v* is in S.**

☞ **Feasible solution set $\mathcal{FS}$: all the vertex covers. $V \in \mathcal{FS}$**

☞ **cost(S) = | S |**

☞ **S ~ S': S' can be obtained from S by (adding or)**

***deleting a single node*.**

〖Example〗 The Vertex Cover Problem.

❖ Vertex cover problem: Given an undirected graph G = (V, E) and an integer K, does G contain a subset V' ⊆ V such that |V'| is (at most) K and every edge in G has a vertex in V' (*vertex cover*)?

❖ **Vertex cover problem: Given an undirected graph G = (V, E). Find a *minimum* subset S of V such that for each edge (*u*, *v*) in E, either *u* or *v* is in S.**

☞ **Feasible solution set $\mathcal{FS}$: all the vertex covers. $V \in \mathcal{FS}$**

☞ **cost(S) = | S |**

☞ **S ~ S': S' can be obtained from S by (adding or)**

***deleting a single node*.**

Each vertex cover S has at most |V| neighbors.

〖Example〗 The Vertex Cover Problem.

❖ Vertex cover problem: Given an undirected graph G = (V, E) and an integer K, does G contain a subset V' ⊆ V such that |V'| is (at most) K and every edge in G has a vertex in V' (*vertex cover*)?

❖ **Vertex cover problem: Given an undirected graph G = (V, E). Find a *minimum* subset S of V such that for each edge (*u*, *v*) in E, either *u* or *v* is in S.**

☞ **Feasible solution set $\mathcal{FS}$: all the vertex covers. $V \in \mathcal{FS}$**

☞ **cost(S) = | S |**

☞ **S ~ S': S' can be obtained from S by (adding or)**

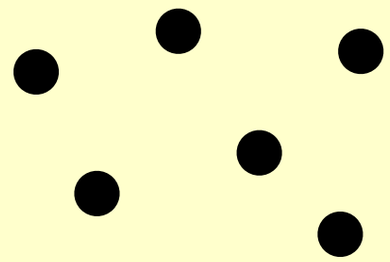
***deleting a single node*.**

Each vertex cover S has at most |V| neighbors.

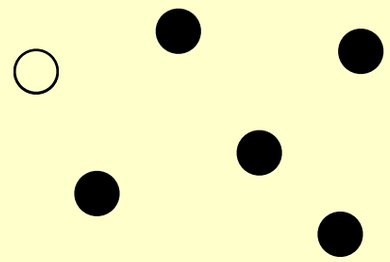
☞ **Search: Start from S = V; delete a node and check if S' is a vertex cover with a smaller cost.**

Case 0:

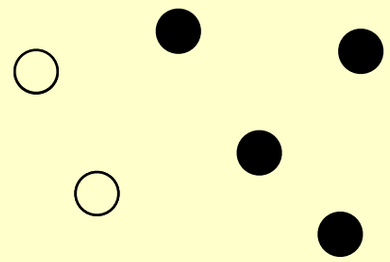
Case 0:



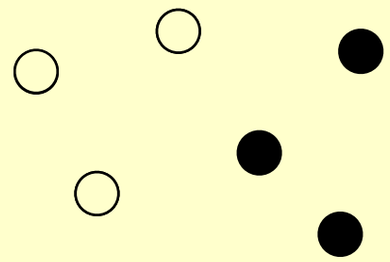
Case 0:



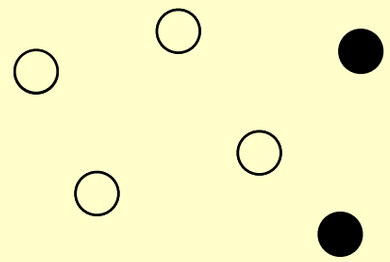
Case 0:



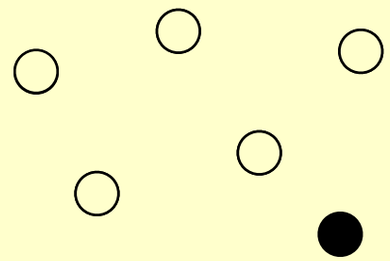
Case 0:



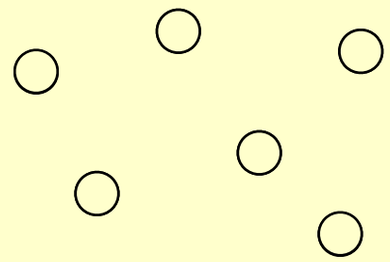
Case 0:



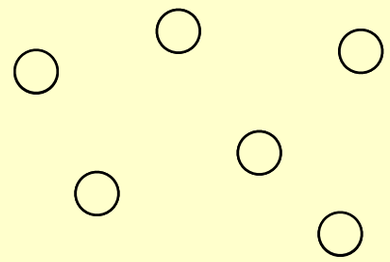
Case 0:

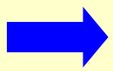


Case 0:



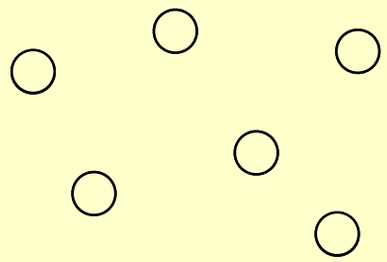
Case 0:





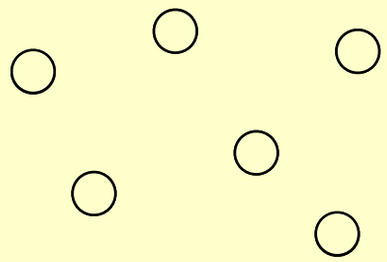
S = ∅

Case 0:

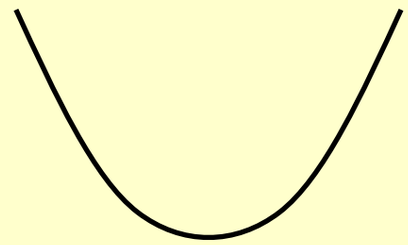


S = ∅

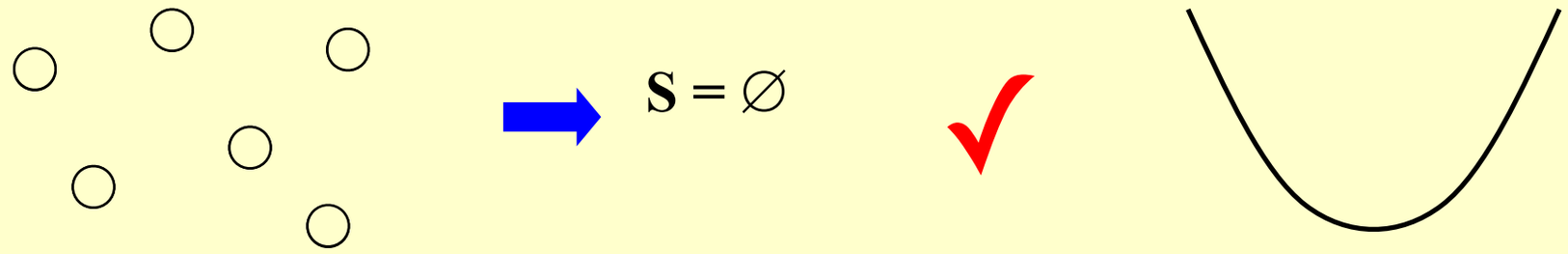
Case 0:



$\mathbf{S} = \varnothing$

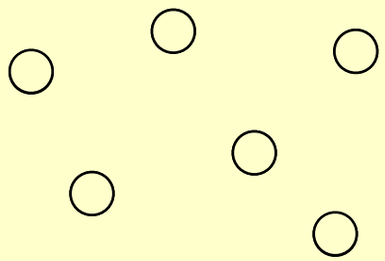


Case 0:

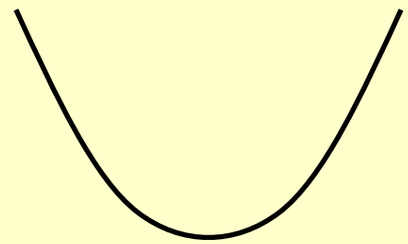


Case 1:

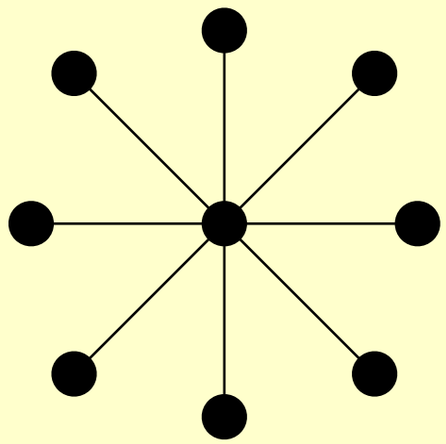
Case 0:



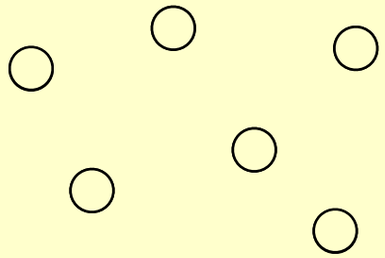
$S = \varnothing$



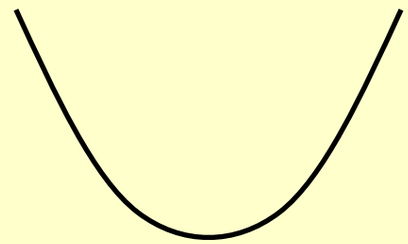
Case 1:



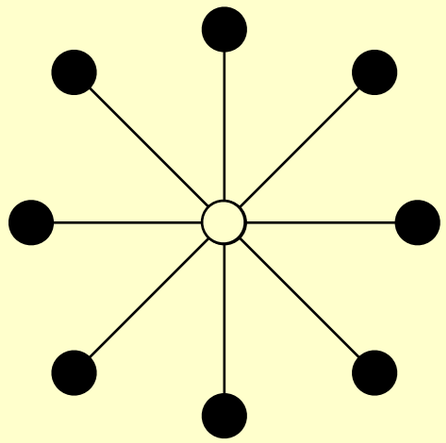
Case 0:



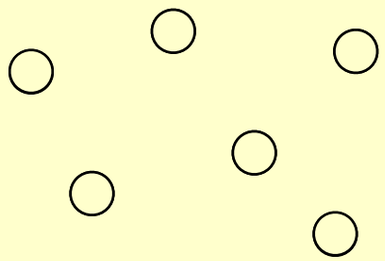
$S = \varnothing$



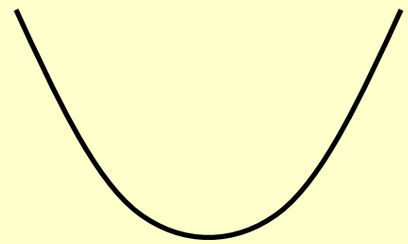
Case 1:



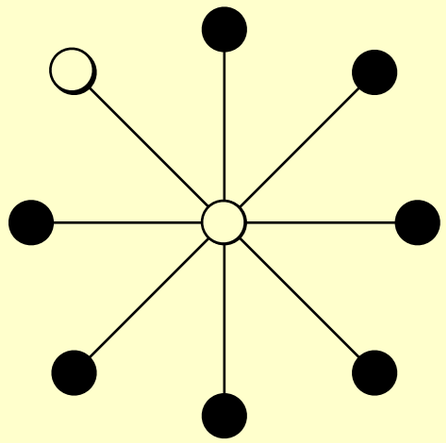
Case 0:



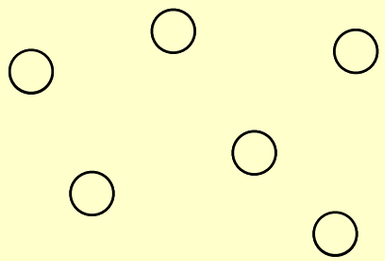
$S = \varnothing$



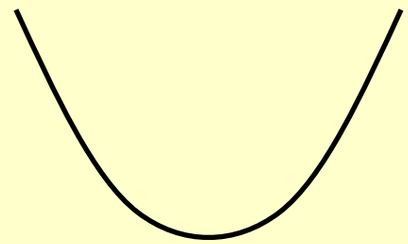
Case 1:



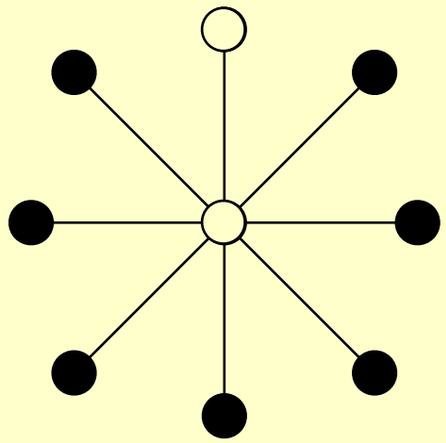
Case 0:



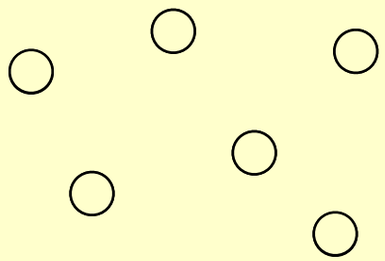
$S = \varnothing$



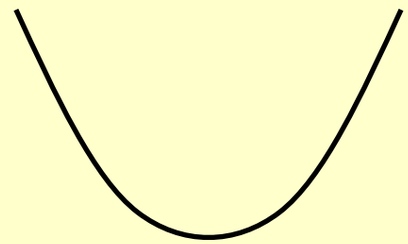
Case 1:



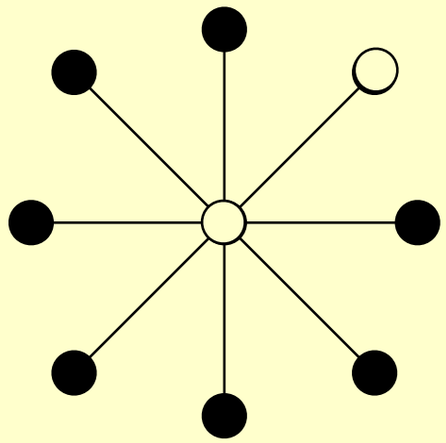
Case 0:



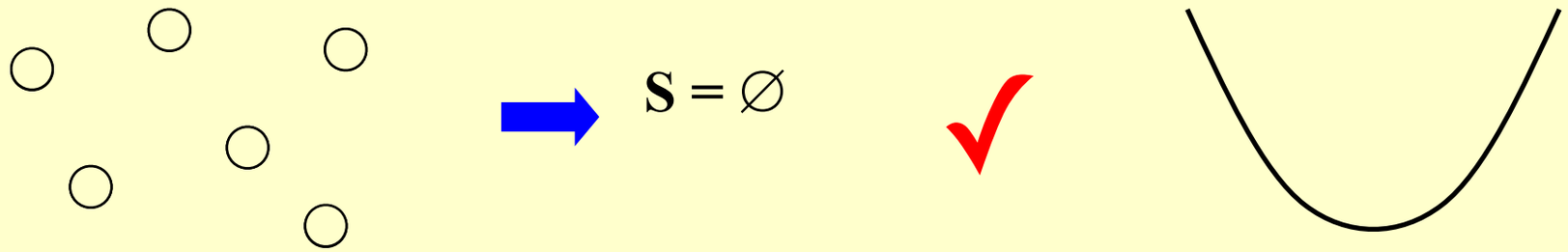
$S = \varnothing$



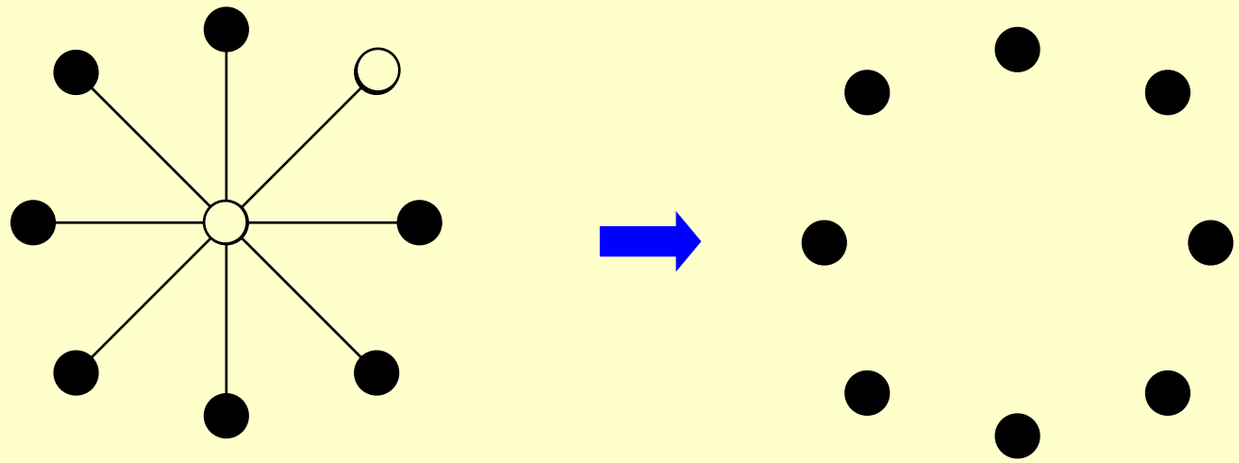
Case 1:



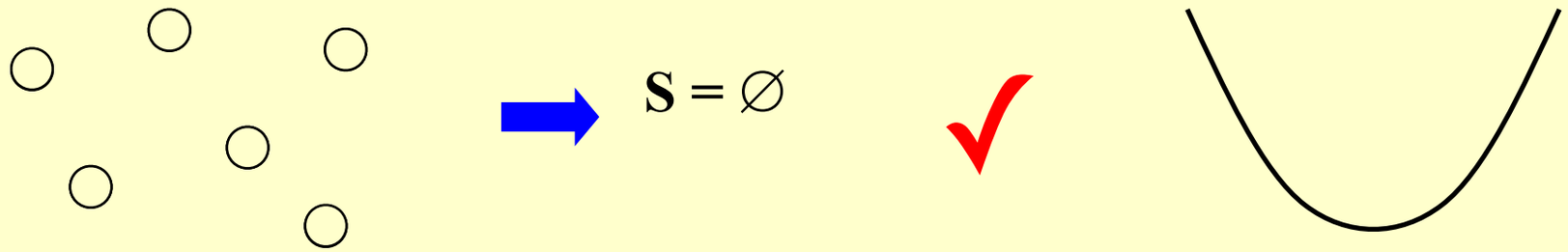
Case 0:



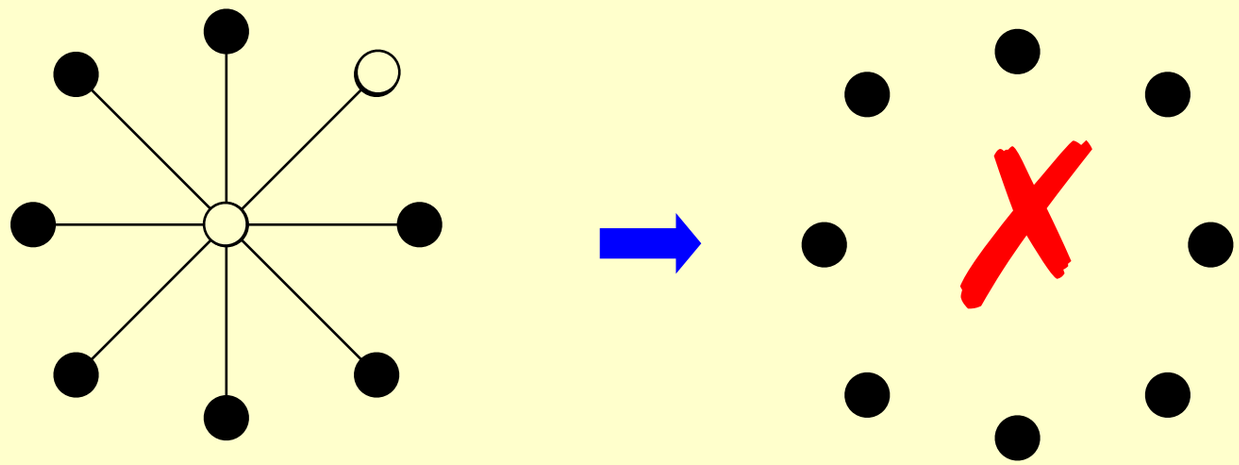
Case 1:



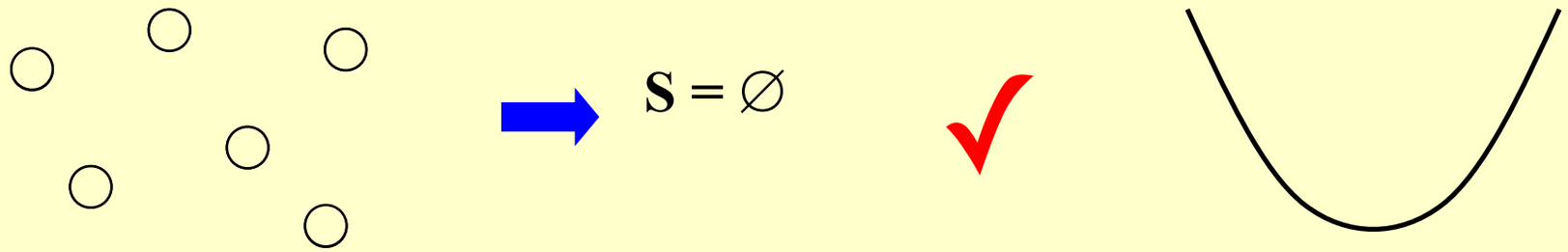
Case 0:



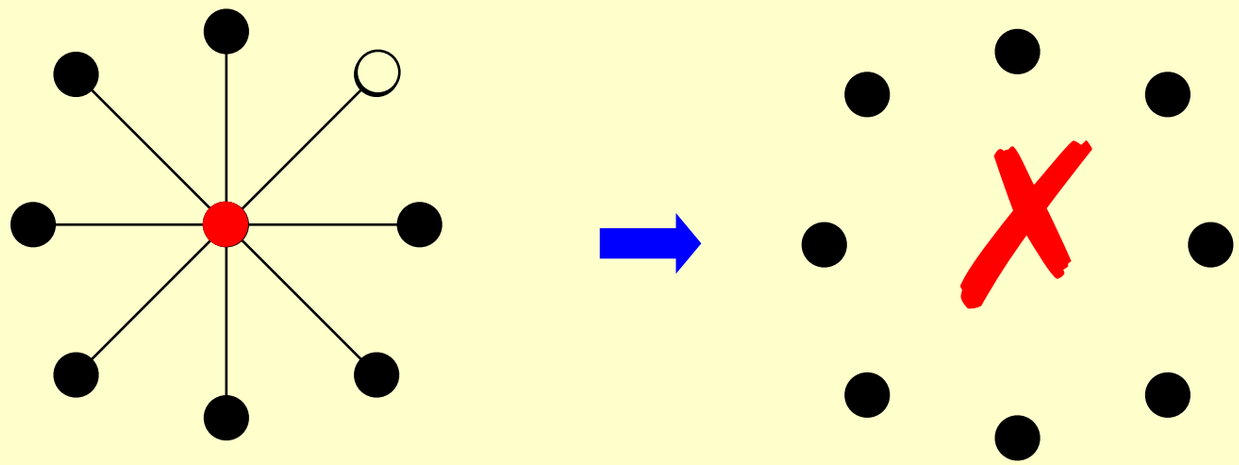
Case 1:



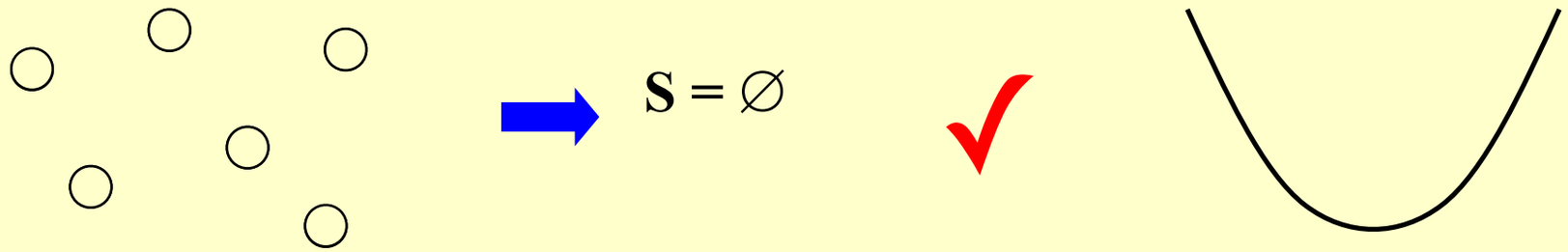
Case 0:



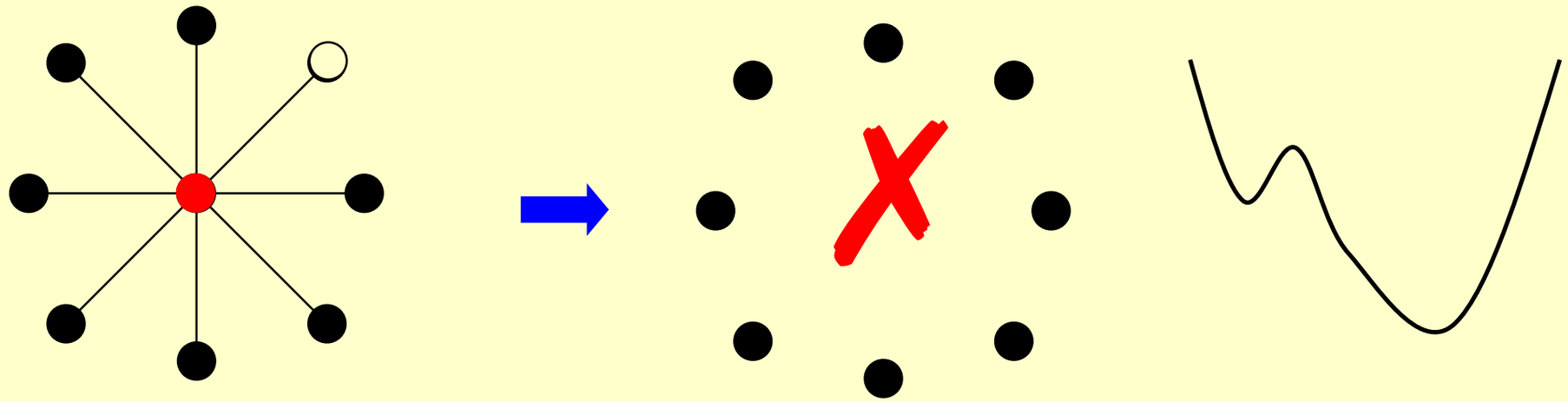
Case 1:



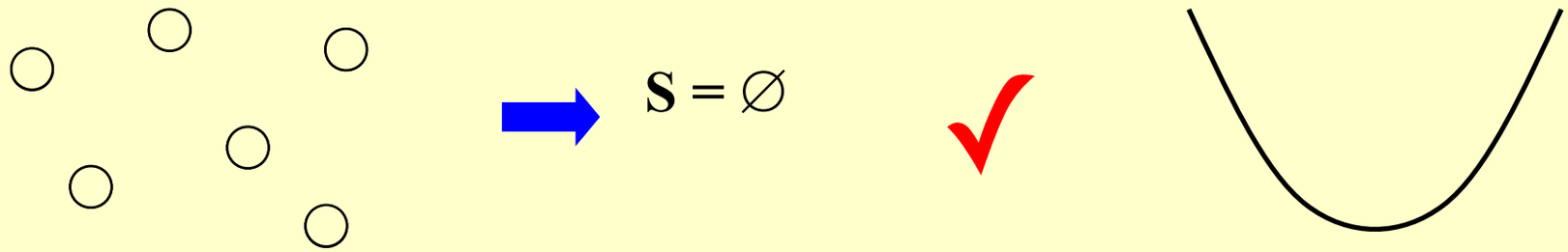
Case 0:



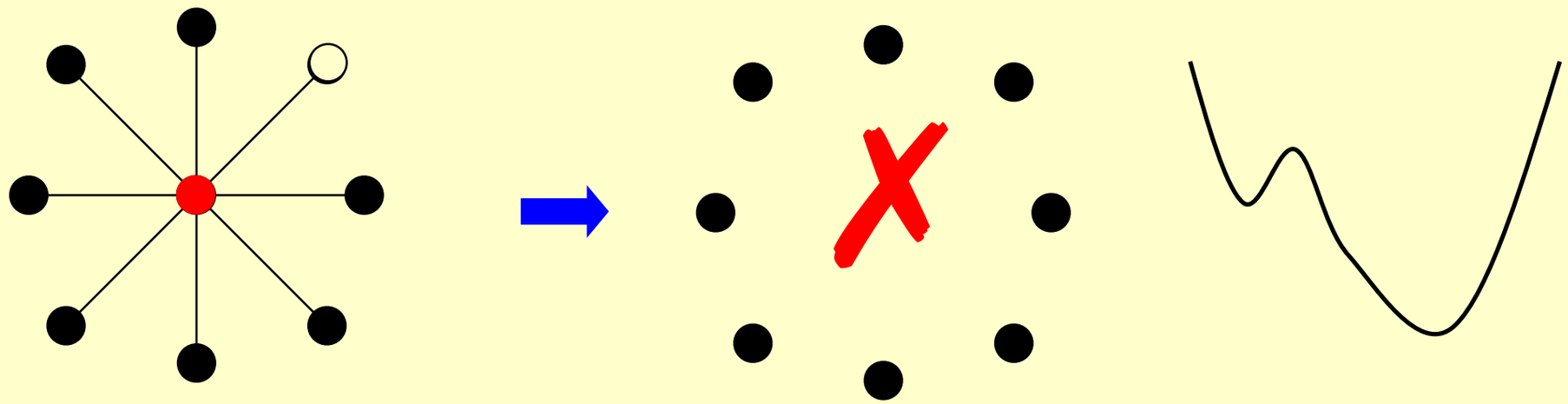
Case 1:



Case 0:

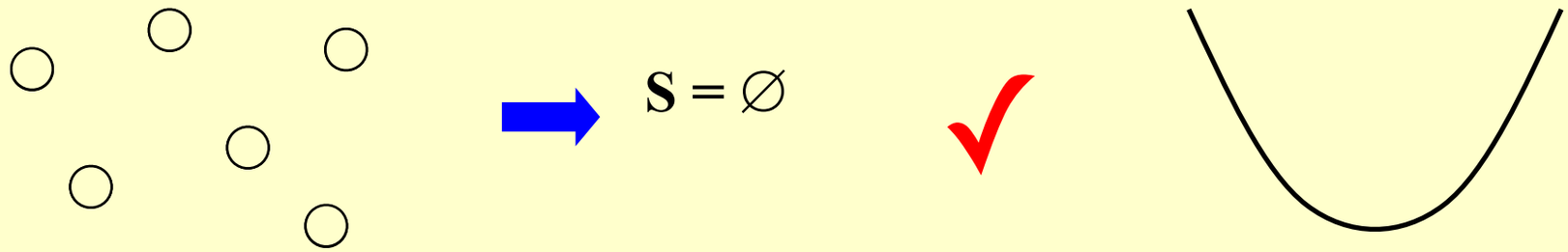


Case 1:

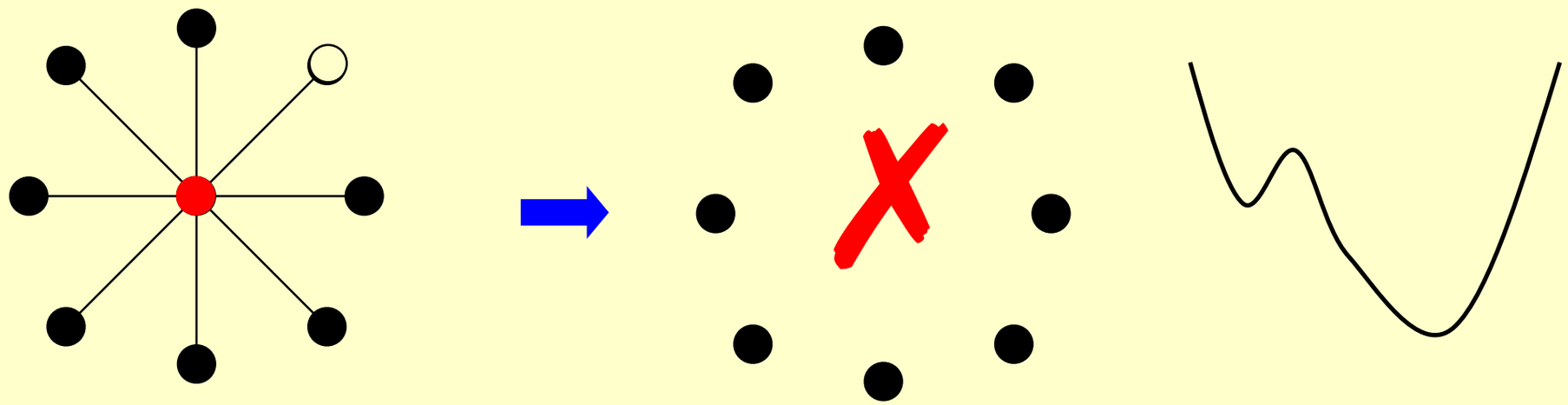


Case 2:

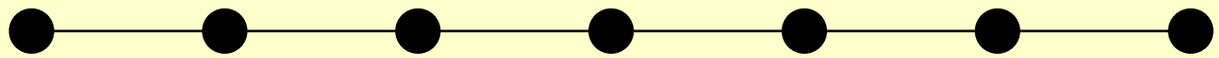
Case 0:



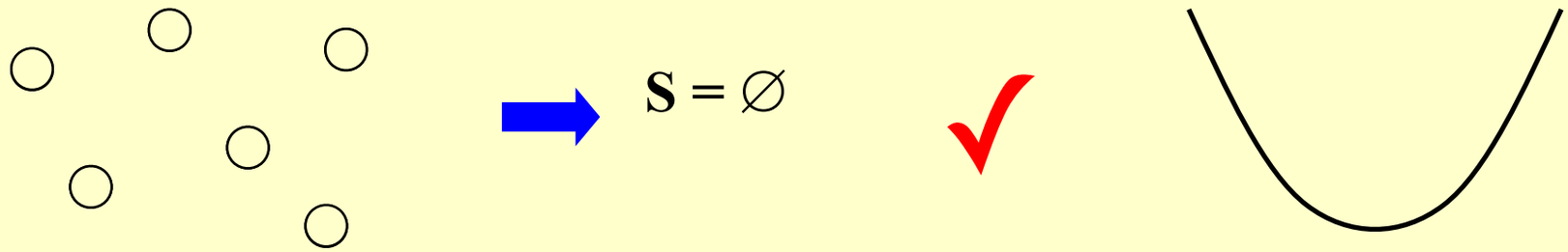
Case 1:



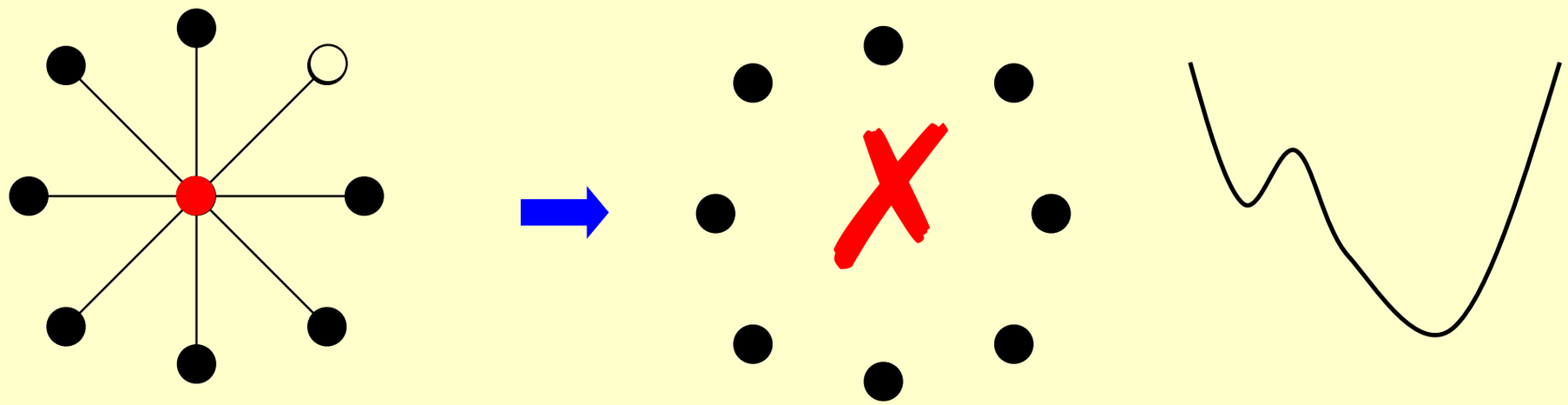
Case 2:



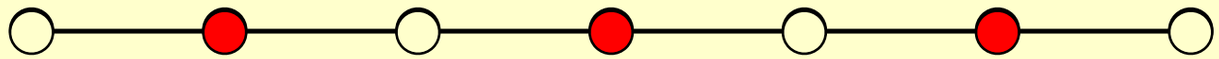
Case 0:



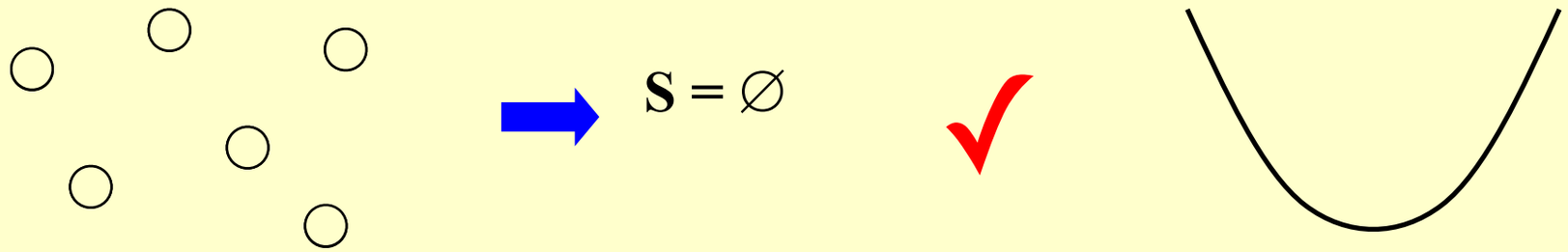
Case 1:



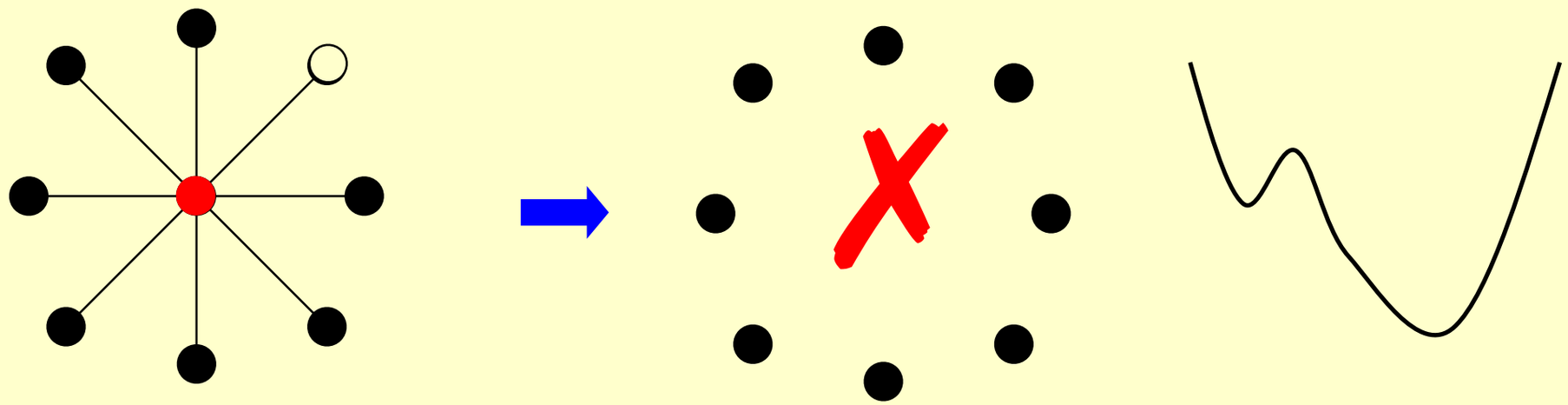
Case 2:



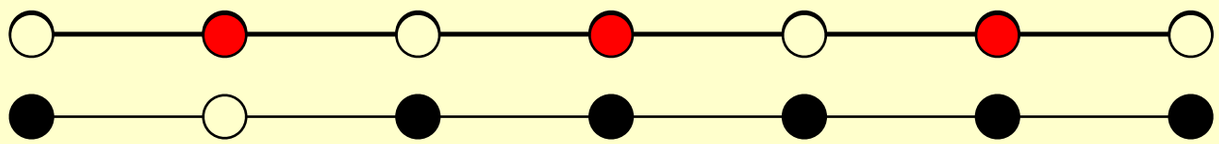
Case 0:



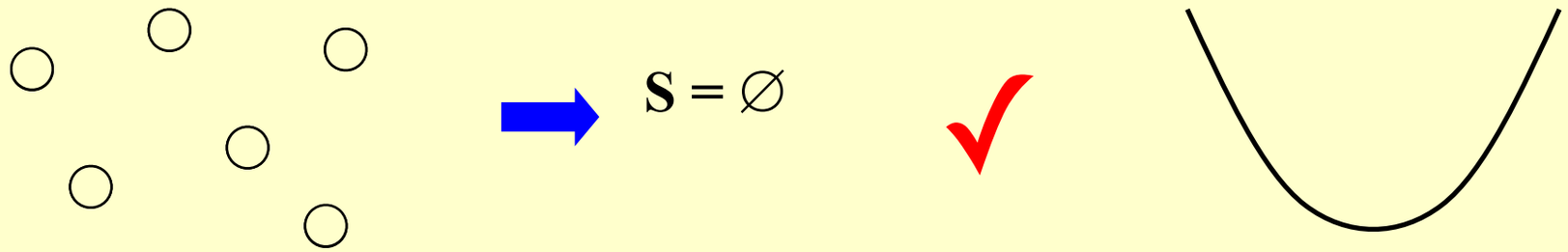
Case 1:



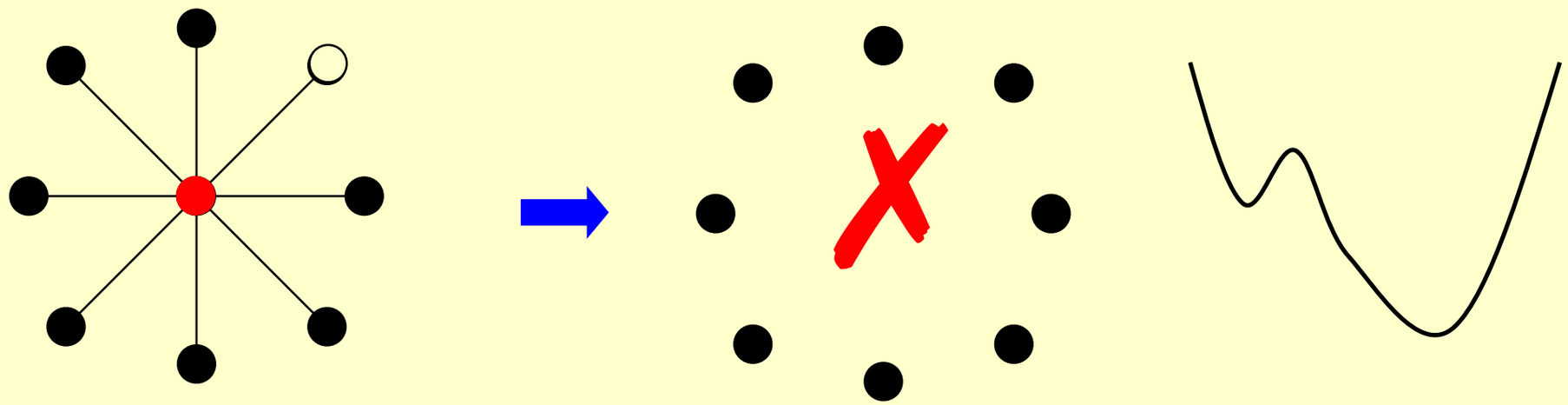
Case 2:



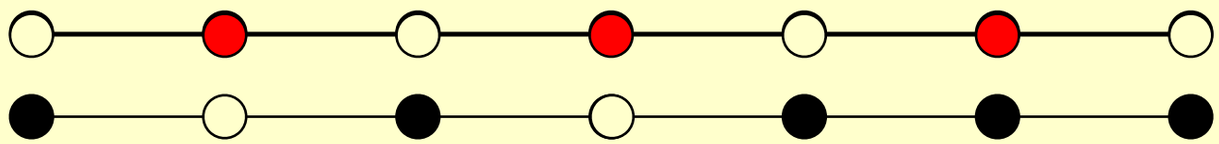
Case 0:



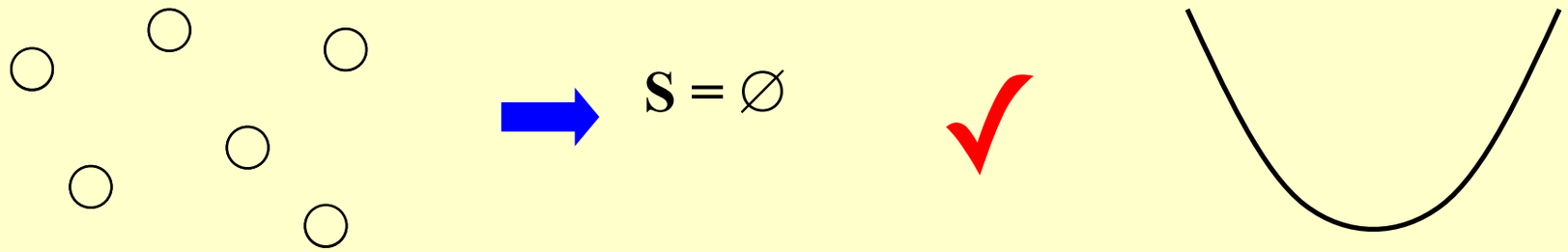
Case 1:



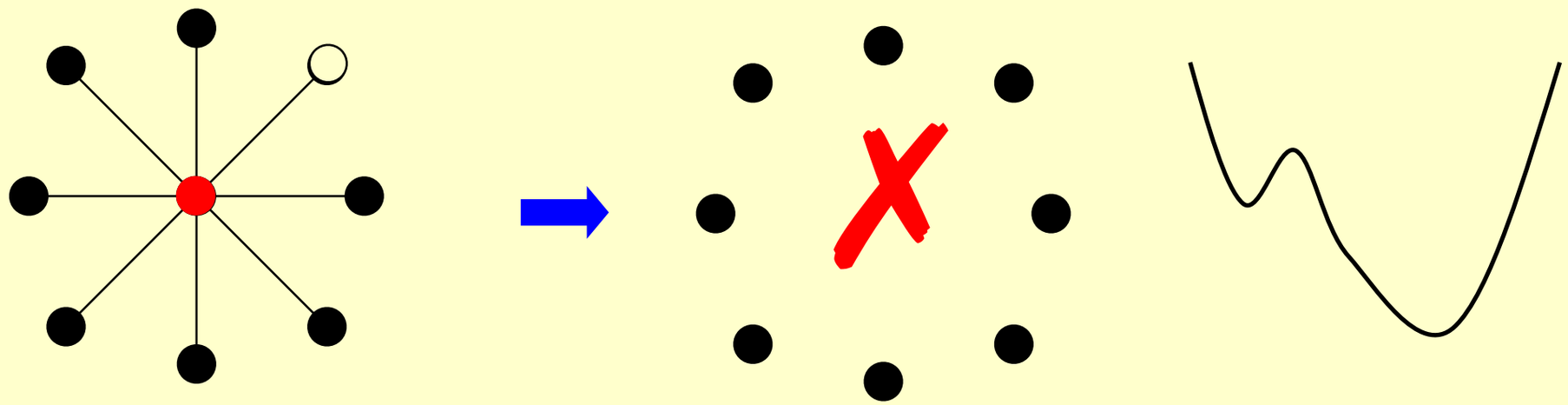
Case 2:



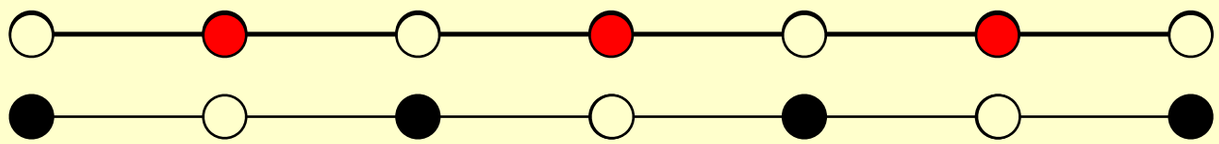
Case 0:



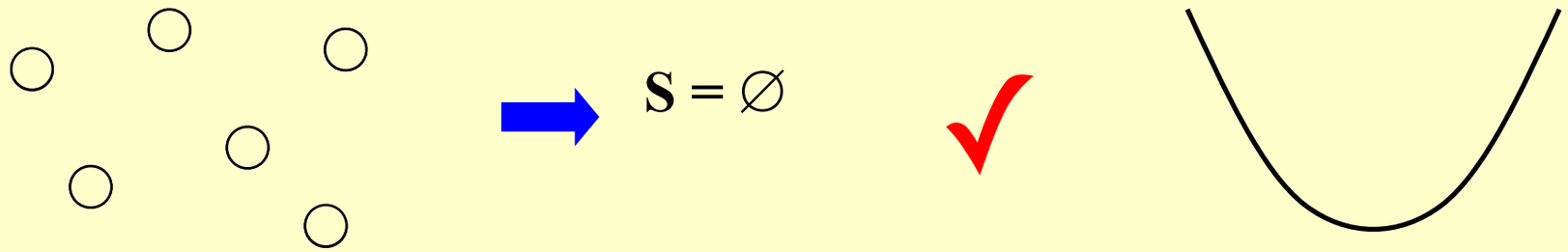
Case 1:



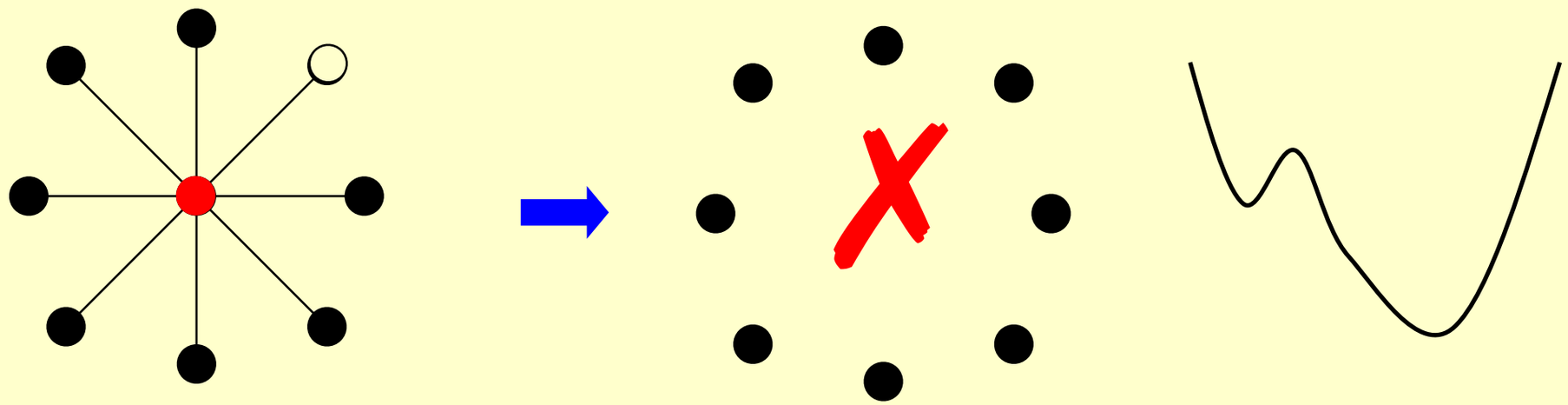
Case 2:



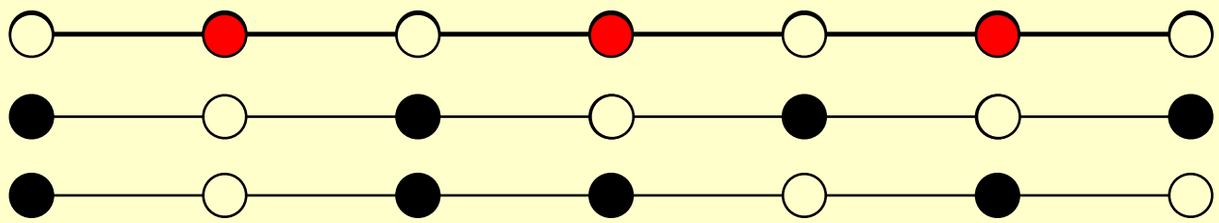
Case 0:



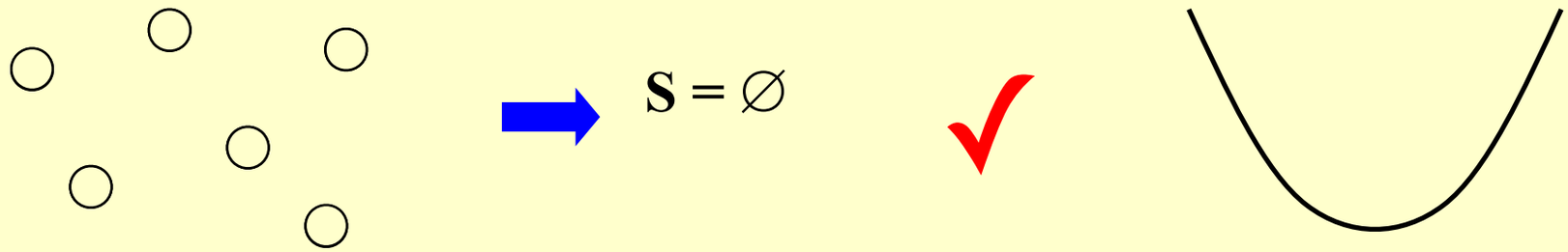
Case 1:



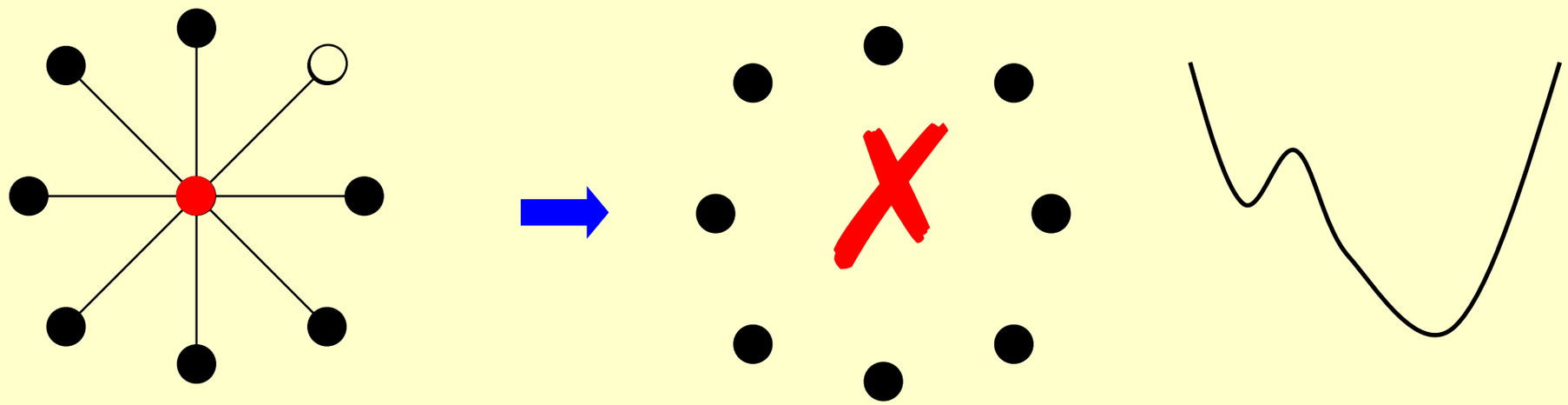
Case 2:



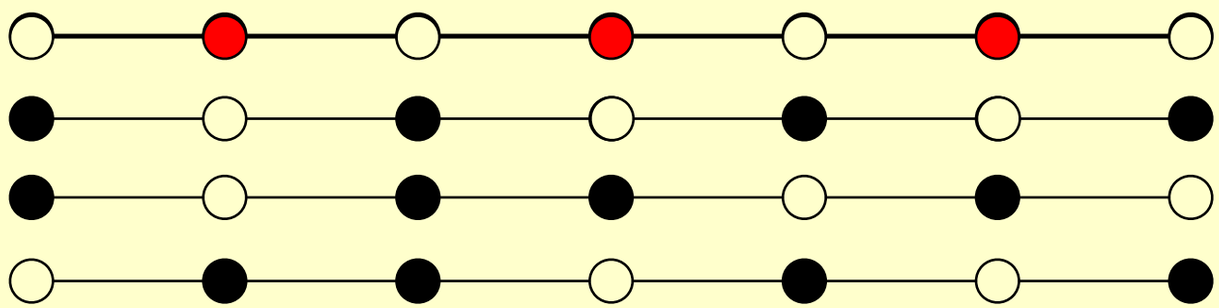
Case 0:



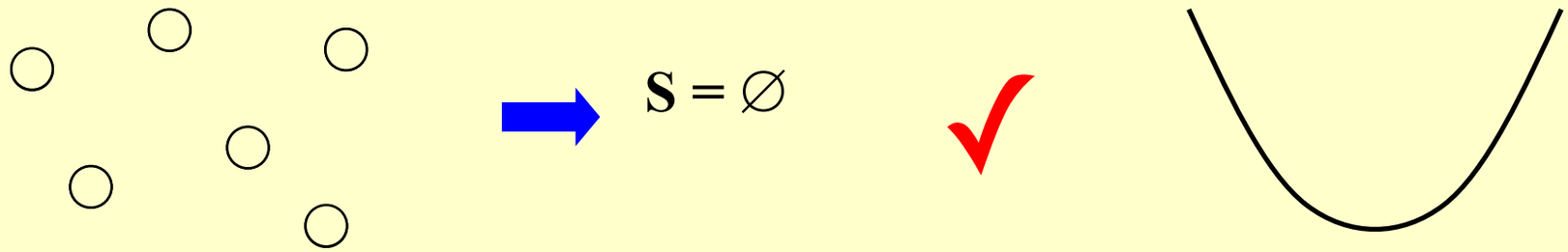
Case 1:



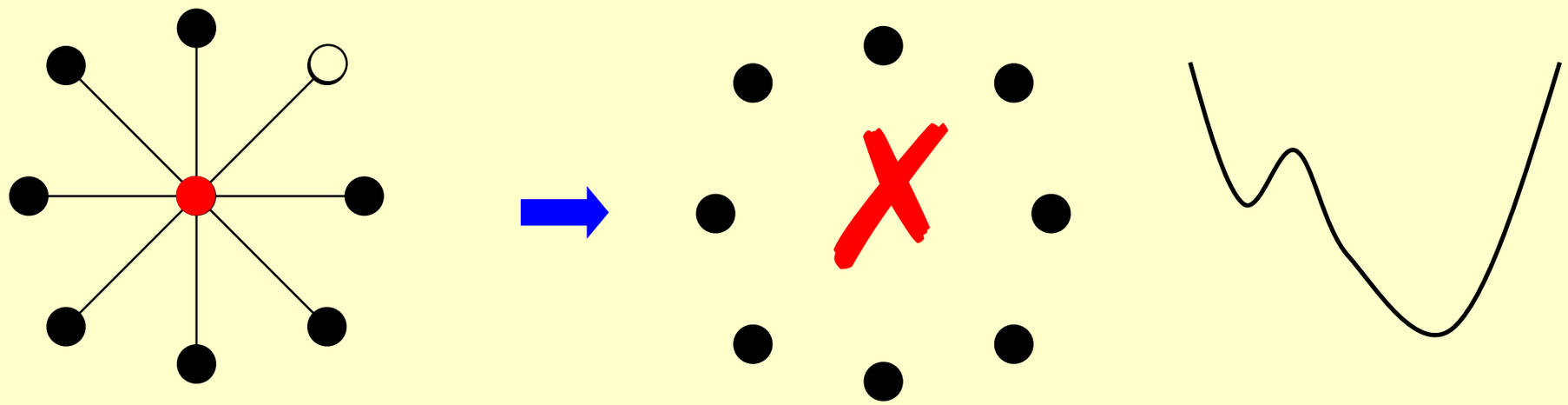
Case 2:



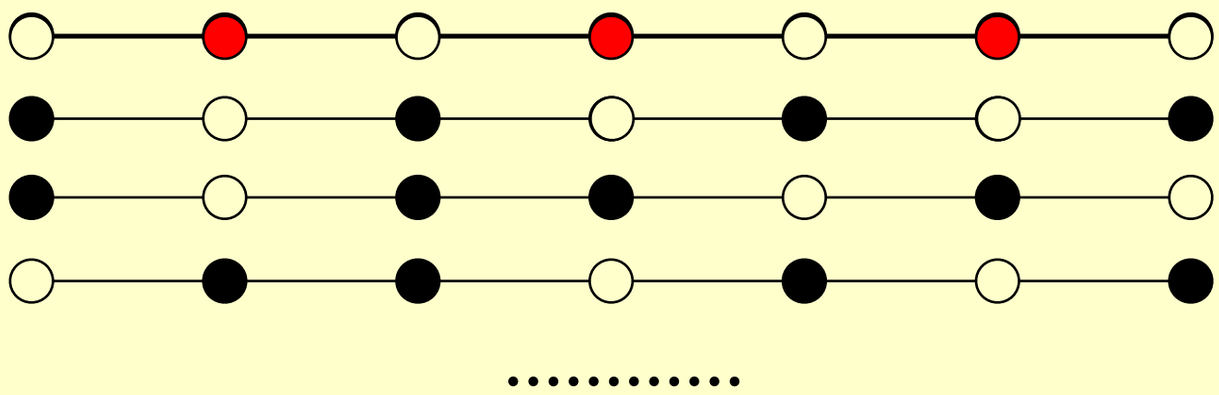
Case 0:



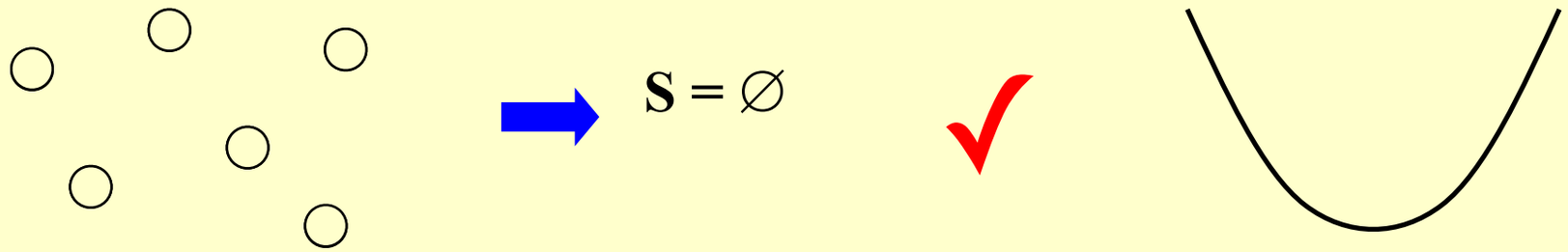
Case 1:



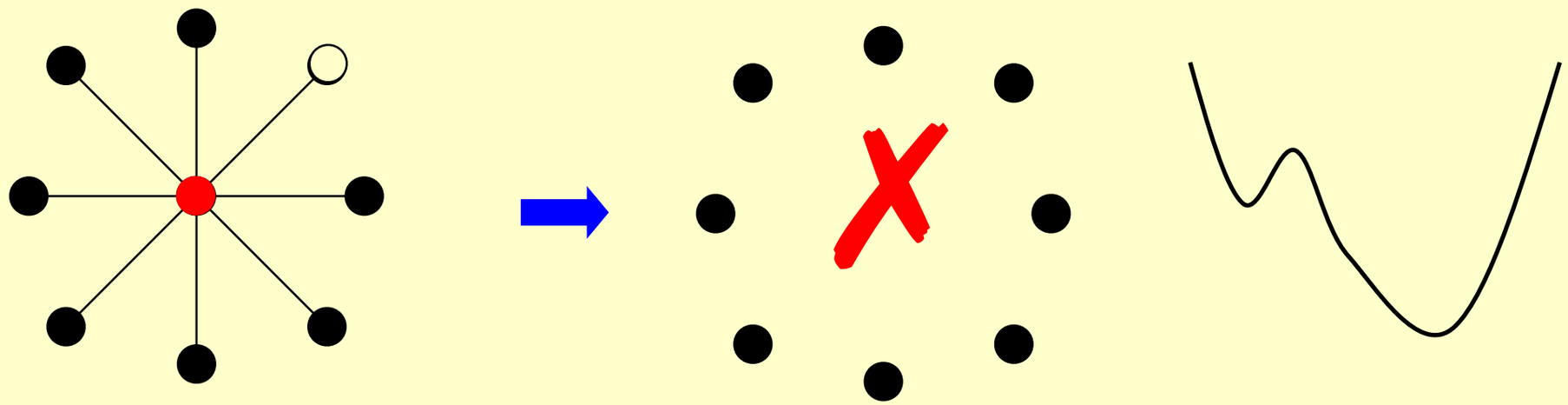
Case 2:



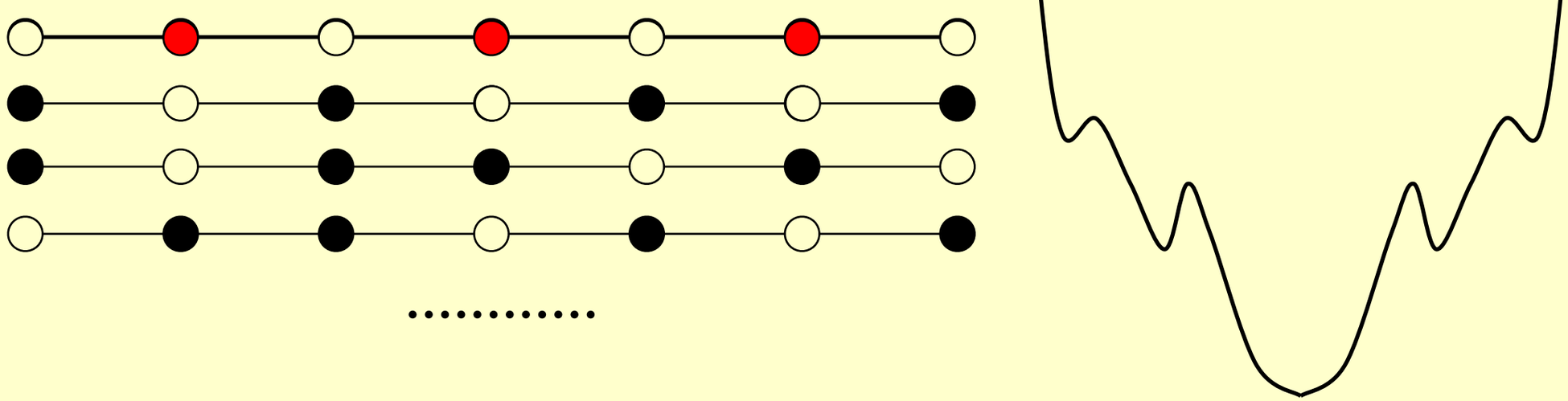
Case 0:



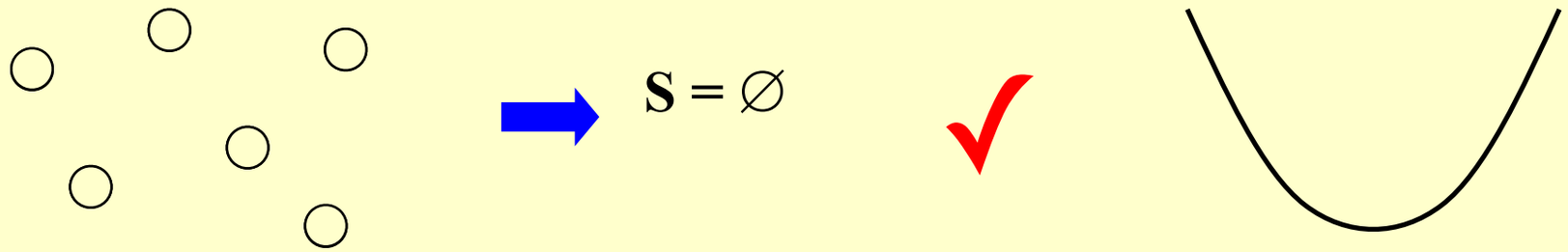
Case 1:



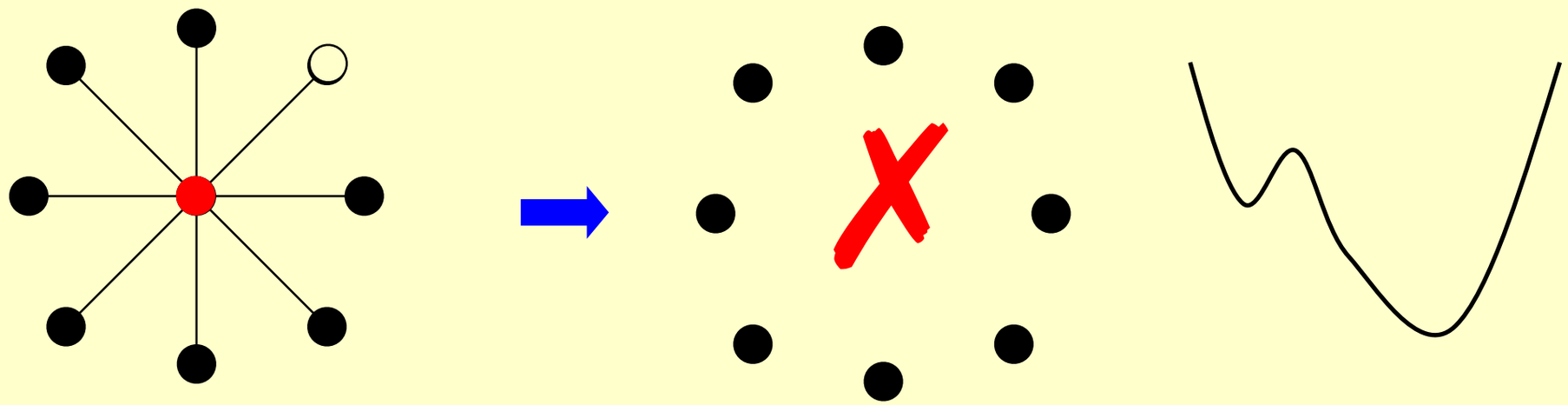
Case 2:



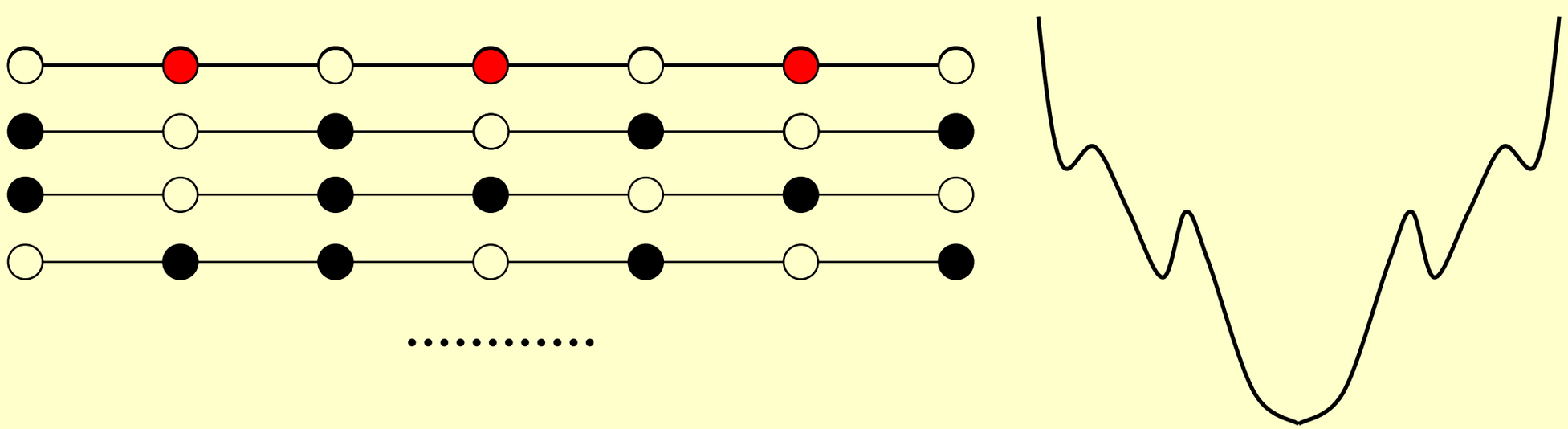
Case 0:



Case 1:



Case 2:



Discussion 17:

Can you give another case in which gradient descent doesn't work?

Outline: Local Search

- Gradient descent
- Metropolis algorithm and simulated annealing
- Hopfield neural networks
- Max-cut problem
- Take-home messages

Metropolis algorithm

Metropolis algorithm.

- Simulate behavior of a physical system according to principles of statistical mechanics.
- Globally biased toward "downhill" steps, but occasionally makes "uphill" steps to break out of local minima.

THE JOURNAL OF CHEMICAL PHYSICS VOLUME 21, NUMBER 6 JUNE, 1953

Equation of State Calculations by Fast Computing Machines

NICHOLAS METROPOLIS, ARIANNA W. ROSENBLUTH, MARSHALL N. ROSENBLUTH, AND AUGUSTA H. TELLER,
Los Alamos Scientific Laboratory, Los Alamos, New Mexico

AND

EDWARD TELLER,* *Department of Physics, University of Chicago, Chicago, Illinois*

(Received March 6, 1953)

A general method, suitable for fast computing machines, for investigating such properties as equations of state for substances consisting of interacting individual molecules is described. The method consists of a modified Monte Carlo integration over configuration space. Results for the two-dimensional rigid-sphere system have been obtained on the Los Alamos MANIAC and are presented here. These results are compared to the free volume equation of state and to a four-term virial coefficient expansion.

Gibbs-Boltzmann function

Gibbs-Boltzmann function. The probability of finding a physical system in a state with energy E is proportional to $e^{-E/(kT)}$, where $T > 0$ is temperature and k is a constant.

- For any temperature $T > 0$, function is monotone decreasing function of energy E.
- System more likely to be in a lower energy state than higher one.
  - T large: high and low energy states have roughly same probability
  - T small: low energy states are much more probable

Metropolis algorithm

Metropolis algorithm.

- Given a fixed temperature T, maintain current state S.
- Randomly perturb current state S to new state $S' \in N(S)$.
- If $E(S') \leq E(S)$, update current state to S'.

  Otherwise, update current state to S' with probability $e^{-\Delta E / (kT)}$,

  where $\Delta E = E(S') - \mathrm{E(S)} > 0$.

Metropolis algorithm

have some probability to step back.

Metropolis algorithm.

- Given a fixed temperature T, maintain current state S.
- Randomly perturb current state S to new state $S' \in N(S)$.
- If $E(S') \leq E(S)$, update current state to S'.
  Otherwise, update current state to S' with probability $e^{-\Delta E / (kT)}$,
  where $\Delta E = E(S') - \mathrm{E(S)} > 0$.

Metropolis algorithm

have some probability to step back.

Metropolis algorithm.

- Given a fixed temperature T, maintain current state S.
- Randomly perturb current state S to new state $S' \in N(S)$.
- If $E(S') \leq E(S)$, update current state to S'.
  Otherwise, update current state to S' with probability $e^{-\Delta E / (kT)}$,
  where $\Delta E = E(S') - \mathrm{E(S)} > 0$.

Theorem. Let $f_S(t)$ be fraction of first t steps in which simulation is in state S. Then, assuming some technical conditions, with probability 1:

$$\lim_{t \to \infty} f_S(t) = \frac{1}{Z} e^{-E(S)/(kT)},$$

$$\text{where } Z = \sum_{S \in N(S)} e^{-E(S)/(kT)}.$$

Intuition. Simulation spends roughly the right amount of time in each state, according to Gibbs-Boltzmann equation.

Simulated annealing

Simulated annealing.

- T large $\Rightarrow$ probability of accepting an uphill move is large.
- T small $\Rightarrow$ uphill moves are almost never accepted.
- Idea: turn knob to control T.
- Cooling schedule: $T = T(i)$ at iteration i.

Physical analog.

- Take solid and raise it to high temperature, we do not expect it to maintain a nice crystal structure.
- Take a molten solid and freeze it very abruptly, we do not expect to get a perfect crystal either.
- Annealing: cool material gradually from high temperature, allowing it to reach equilibrium at succession of intermediate lower temperatures.

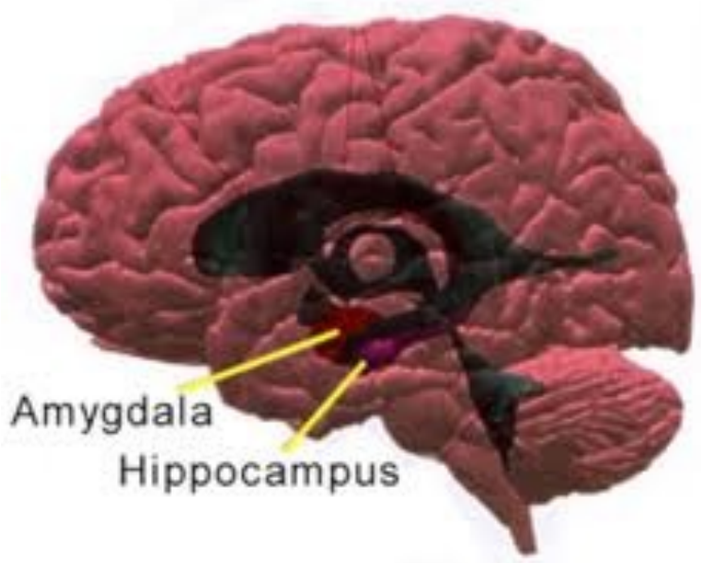
Outline: Local Search

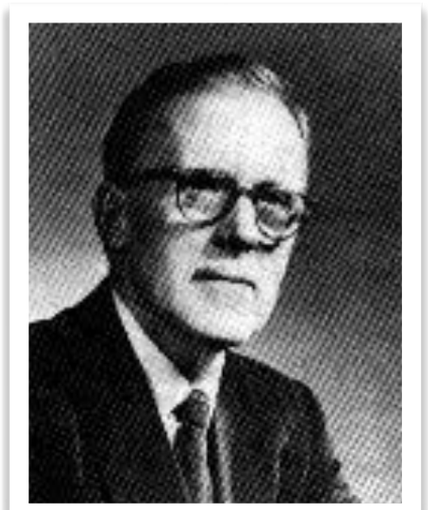
- Gradient descent
- Metropolis algorithm and simulated annealing
- Hopfield neural networks
- Max-cut problem
- Take-home messages

Hopfield Neural Networks

Human learning

- Learning is to associate two events with each other.
- In the Hebbian type of learning, both presynaptic and postsynaptic neurons are involved.
- The main brain organ for learning/explicit memory is the hippocampus (of the limbic system) using Hebbian type.

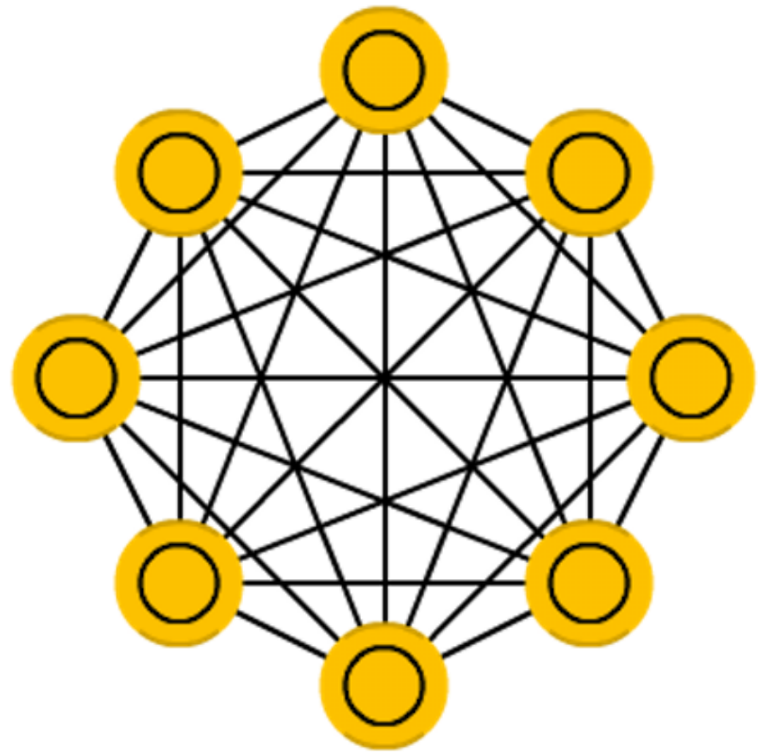




Donald O. Hebb

Hopfield Neural Networks

- Fully-connect undirected graphs.
- each node represents binary states -1 or +1.
- each edge has a weight representing the strength of association between two end nodes.



Hopfield Neural Networks

moscow------russia
lima----------peru
london-----england
tokyo--------japan
edinburgh-scotland
ottawa------canada
oslo--------norway
stockholm---sweden
paris-------france

desired associative memory

moscow---::::::::: $\Longrightarrow$ moscow------russia

::::::::::--canada $\Longrightarrow$ ottawa------canada

functionality 1: pattern completion

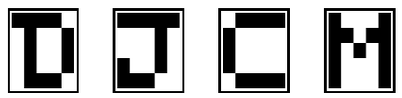
otowa-------canada $\Longrightarrow$ ottawa------canada

egindurrh-sxotland $\Longrightarrow$ edinburgh-scotland

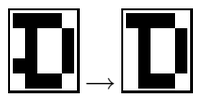
functionality 2: error correction

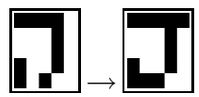
Hopfield Neural Networks

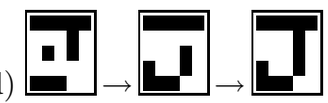
(a)

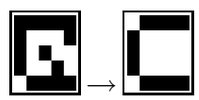


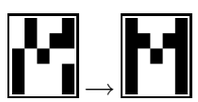
.	0	0	0	0	-2	2	-2	2	2	-2	0	0	0	2	0	0	-2	0	2	2	0	0	-2	-2
0	.	4	4	0	-2	-2	-2	-2	-2	-2	0	-4	0	-2	0	0	-2	0	-2	-2	4	4	2	-2
0	4	.	4	0	-2	-2	-2	-2	-2	-2	0	-4	0	-2	0	0	-2	0	-2	-2	4	4	2	-2
0	4	4	.	0	-2	-2	-2	-2	-2	-2	0	-4	0	-2	0	0	-2	0	-2	-2	4	4	2	-2
0	0	0	0	.	2	-2	-2	2	-2	2	-4	0	0	-2	4	-4	-2	0	-2	2	0	0	-2	2
-2	-2	-2	-2	2	.	0	0	0	0	4	-2	2	-2	0	2	-2	0	-2	0	0	-2	-2	0	4
2	-2	-2	-2	-2	0	.	0	0	4	0	2	2	-2	4	-2	2	0	-2	4	0	-2	-2	0	0
-2	-2	-2	-2	-2	0	0	.	0	0	0	2	2	2	0	-2	2	4	2	0	0	-2	-2	0	0
2	-2	-2	-2	2	0	0	0	.	0	0	-2	2	2	0	2	-2	0	2	0	4	-2	-2	-4	0
2	-2	-2	-2	-2	0	4	0	0	.	0	2	2	-2	4	-2	2	0	-2	4	0	-2	-2	0	0
-2	-2	-2	-2	2	4	0	0	0	0	.	-2	2	-2	0	2	-2	0	-2	0	0	-2	-2	0	4
0	0	0	0	-4	-2	2	2	-2	2	-2	.	0	0	2	-4	4	2	0	2	-2	0	0	2	-2
0	-4	-4	-4	0	2	2	2	2	2	2	0	.	0	2	0	0	2	0	2	2	-4	-4	-2	2
0	0	0	0	0	-2	-2	2	2	-2	-2	0	0	.	-2	0	0	2	4	-2	2	0	0	-2	-2
2	-2	-2	-2	-2	0	4	0	0	4	0	2	2	-2	.	-2	2	0	-2	4	0	-2	-2	0	0
0	0	0	0	4	2	-2	-2	2	-2	2	-4	0	0	-2	.	-4	-2	0	-2	2	0	0	-2	2
0	0	0	0	-4	-2	2	2	-2	2	-2	4	0	0	2	-4	.	2	0	2	-2	0	0	2	-2
-2	-2	-2	-2	-2	0	0	4	0	0	0	2	2	2	0	-2	2	.	2	0	0	-2	-2	0	0
0	0	0	0	0	-2	-2	2	2	-2	-2	0	0	4	-2	0	0	2	.	-2	2	0	0	-2	-2
2	-2	-2	-2	-2	0	4	0	0	4	0	2	2	-2	4	-2	2	0	-2	.	0	-2	-2	0	0
2	-2	-2	-2	2	0	0	0	4	0	0	-2	2	2	0	2	-2	0	2	0	.	-2	-2	-4	0
0	4	4	4	0	-2	-2	-2	-2	-2	-2	0	-4	0	-2	0	0	-2	0	-2	-2	.	4	2	-2
0	4	4	4	0	-2	-2	-2	-2	-2	-2	0	-4	0	-2	0	0	-2	0	-2	-2	4	.	2	-2
-2	2	2	2	-2	0	0	0	-4	0	0	2	-2	-2	0	-2	2	0	-2	0	-4	2	2	.	0
-2	-2	-2	-2	2	4	0	0	0	0	4	-2	2	-2	0	2	-2	0	-2	0	0	-2	-2	0	.

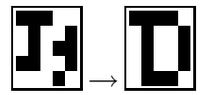
(b) 

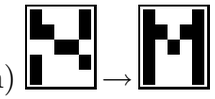
(c) 

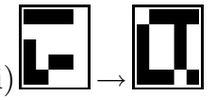
(d) 

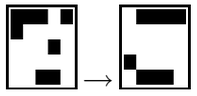
(e) 

(f) 

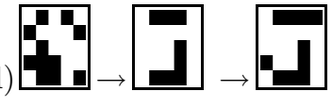
(g) 

(h) 

(i) 

(j) 

(k) 

(l) 

(m) 

Figure 42.3. Binary Hopfield network storing four memories. (a) The four memories, and the weight matrix. (b–h) Initial states that differ by one, two, three, four, or even five bits from a desired memory are restored to that memory in one or two iterations. (i–m) Some initial conditions that are far from the memories lead to stable states other than the four memories; in (i), the stable state looks like a mixture of two memories, 'D' and 'J'; stable state (j) is like a mixture of 'J' and 'C'; in (k), we find a corrupted version of the 'M' memory (two bits distant); in (l) a corrupted version of 'J' (four bits distant) and in (m), a state which looks spurious until we recognize that it is the inverse of the stable state (l).

More details on Hopfield nets see Mackay book Chap. 42.

Hopfield neural networks

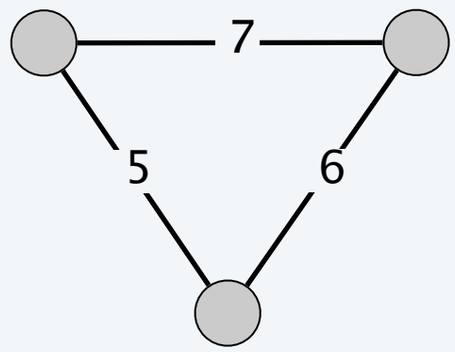
Hopfield networks. Simple model of an associative memory, in which a large collection of units are connected by an underlying network, and neighboring units try to correlate their states.

Input: Graph $G = (V, E)$ with integer (positive or negative) edge weights w.

Configuration. Node assignment $s_u = \pm 1$.

Intuition. If $w_{uv} < 0$, then u and v want to have the same state; if $w_{uv} > 0$ then u and v want different states.

Note. In general, no configuration respects all constraints.



Hopfield neural networks

Hopfield networks. Simple model of an associative memory, in which a large collection of units are connected by an underlying network, and neighboring units try to correlate their states.

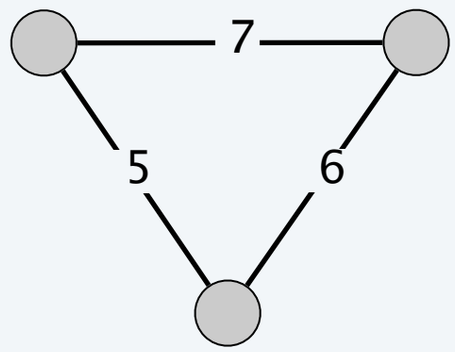
Input: Graph $G = (V, E)$ with integer (positive or negative) edge weights w.

Configuration. Node assignment $s_u = \pm 1$.

Intuition. If $w_{uv} < 0$, then u and v want to have the same state; if $w_{uv} > 0$ then u and v want different states.

Note. In general, no configuration respects all constraints.

In actual Hopfield nets, the situation is the opposite: the end nodes have the same state when the edge weight is positive.



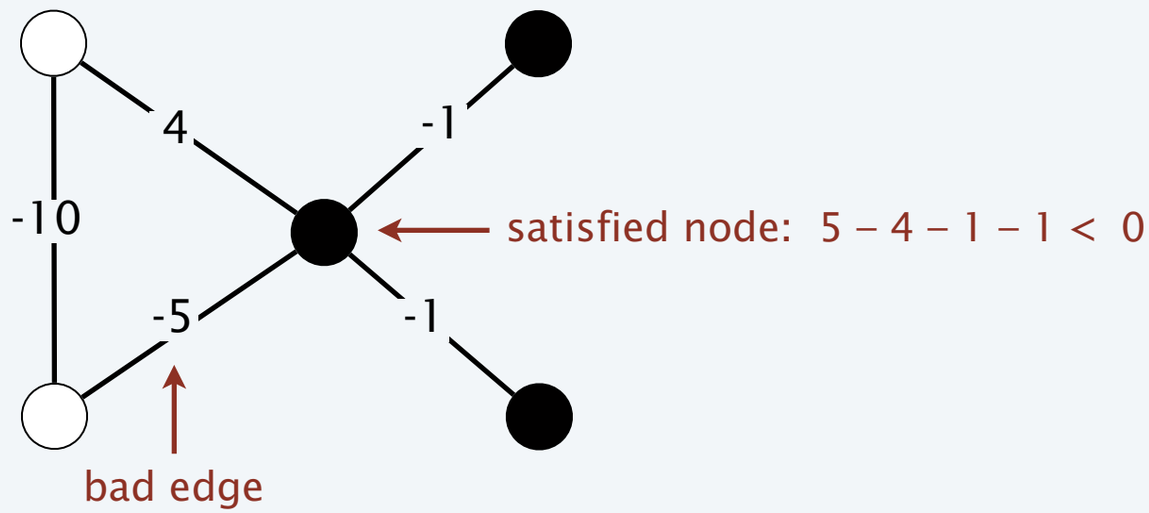
Hopfield neural networks

Def. With respect to a configuration S, edge $e = (u, v)$ is good if $w_e \times s_u \times s_v < 0$. That is, if $w_e < 0$ then $s_u = s_v$; if $w_e > 0$, then $s_u \neq s_v$.

Def. With respect to a configuration S, a node u is satisfied if the weight of incident good edges ≥ weight of incident bad edges.

$$\sum_{v:\ e=(u,v)\in E} w_e\, s_u\, s_v \ \leq\ 0$$

Def. A configuration is stable if all nodes are satisfied.



Goal. Find a stable configuration, if such a configuration exists.

Hopfield neural networks

Goal. Find a stable configuration, if such a configuration exists.

State-flipping algorithm. Repeated flip state of an unsatisfied node.

HOPFIELD-FLIP (G, w)

$S \leftarrow$ *arbitrary configuration*.

WHILE *(current configuration is not stable)*

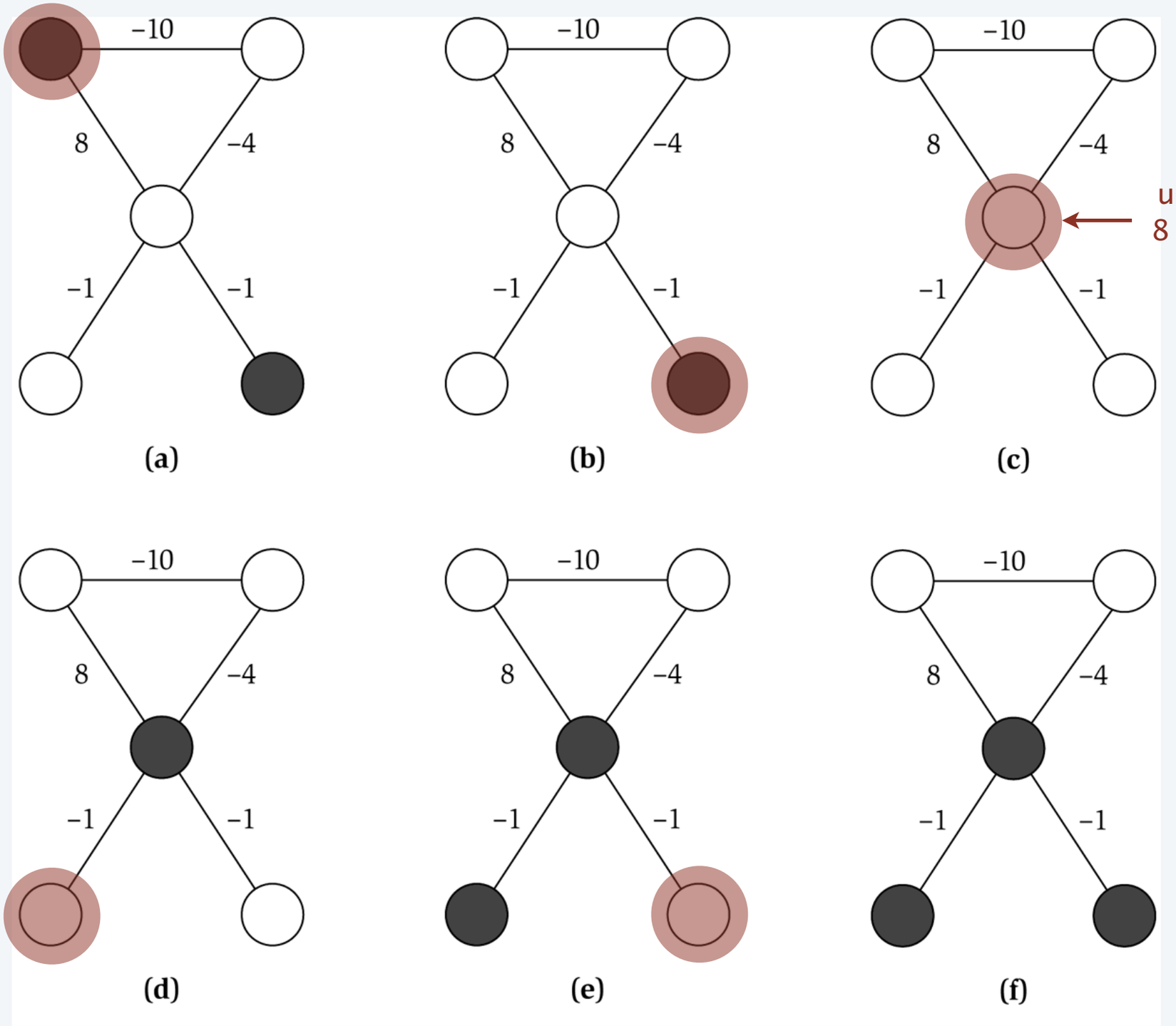
  $u \leftarrow$ *unsatisfied node*.

  $s_u \leftarrow -s_u$.

RETURN S.

State-flipping algorithm example

unsatisfied node
10 − 8 > 0



unsatisfied node
8 − 4 − 1 − 1 > 0

stable

State-flipping algorithm: proof of correctness

Theorem. The state-flipping algorithm terminates with a stable configuration after at most $W = \Sigma_e |w_e|$ iterations.

Pf attempt. Consider measure of progress $\Phi(S)$ = # satisfied nodes.

State-flipping algorithm: proof of correctness

Theorem. The state-flipping algorithm terminates with a stable configuration after at most $W = \Sigma_e |w_e|$ iterations.

Pf. Consider measure of progress $\Phi(S) = \Sigma_{e \text{ good}} |w_e|$.

- Clearly $0 \leq \Phi(S) \leq W$.
- We show $\Phi(S)$ increases by at least 1 after each flip.

  When u flips state:
  - all good edges incident to u become bad
  - all bad edges incident to u become good
  - all other edges remain the same

$$\Phi(S') = \Phi(S) - \sum_{\substack{e:\ e=(u,v)\in E \\ e \text{ is bad}}} |w_e| + \sum_{\substack{e:\ e=(u,v)\in E \\ e \text{ is good}}} |w_e| \geq \Phi(S) + 1$$

u is unsatisfied

Complexity of Hopfield neural network

Hopfield network search problem. Given a weighted graph, find a stable configuration if one exists.

Hopfield network decision problem. Given a weighted graph, does there exist a stable configuration?

Remark. The decision problem is trivially solvable (always yes), but there is no known poly-time algorithm for the search problem.

polynomial in n and log W

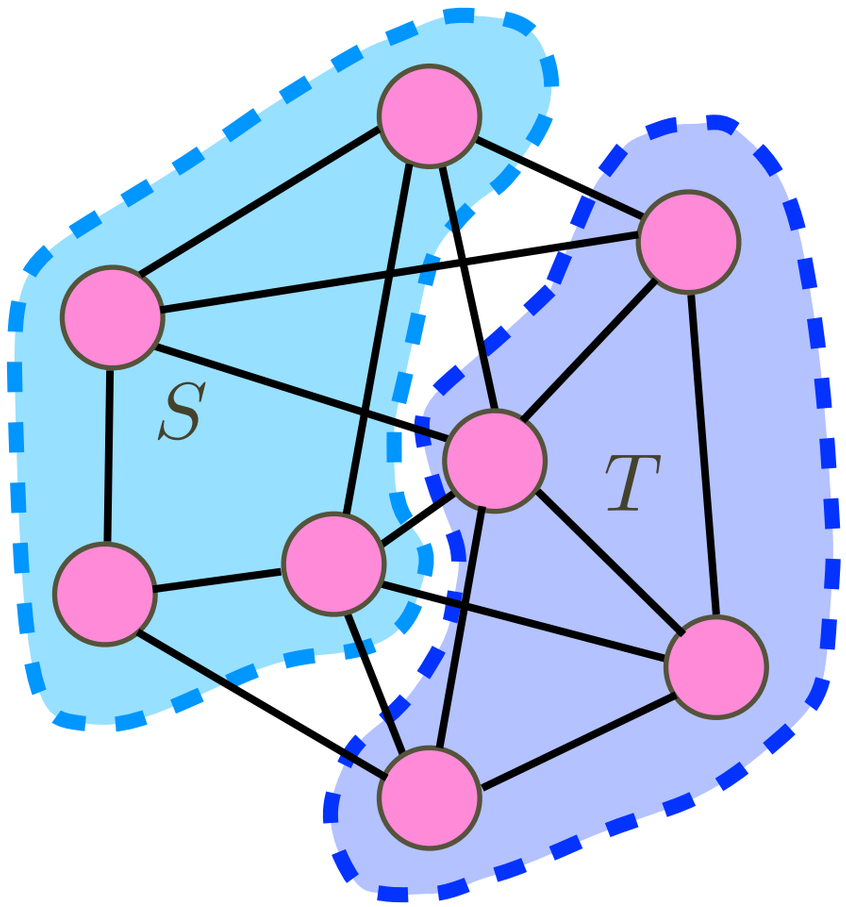
Outline: Local Search

- Gradient descent
- Metropolis algorithm and simulated annealing
- Hopfield neural networks
- Max-cut problem
- Take-home messages

Max-Cut

Instance: An undirected graph $G(V, E)$.

Solution: A bipartition of V into S and T that maximizes the cut $E(S,T) = \{uv \in E: u \in S, v \in T\}$.

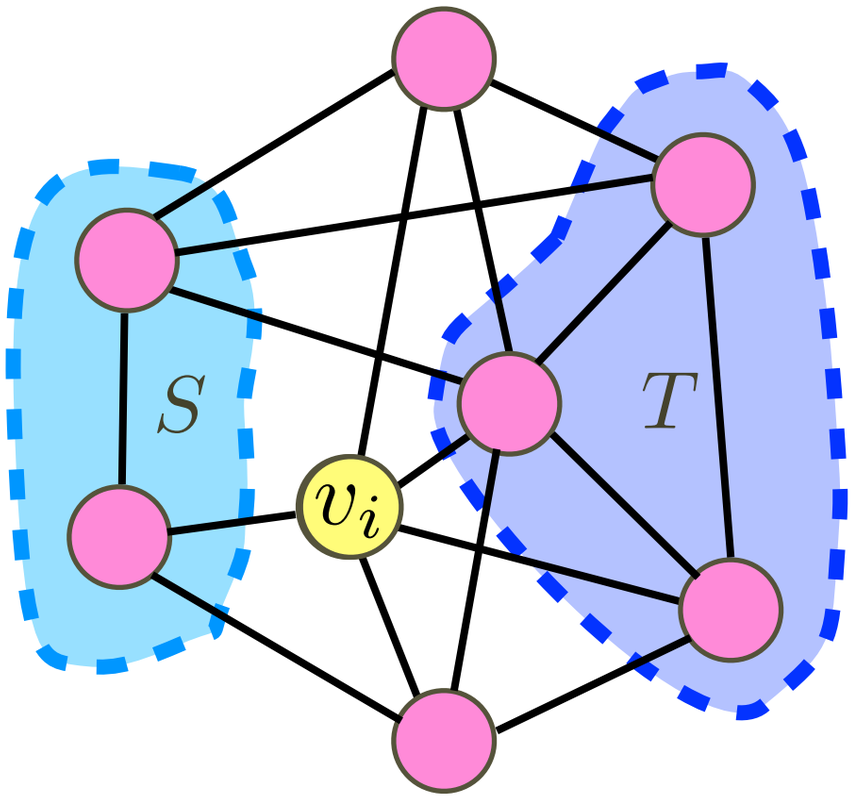


- **NP**-hard.
- One of Karp's 21 **NP**-complete problems (reduction from the *Partition* problem).
- a typical **Max-CSP** (Constraint Satisfaction Problem).
- *Greedy* is 1/2-approximate.

Max-Cut

Instance: An undirected graph $G(V, E)$.

Solution: A bipartition of V into S and T that maximizes the cut $E(S,T) = \{uv \in E: u \in S, v \in T\}$.



GreedyMaxCut

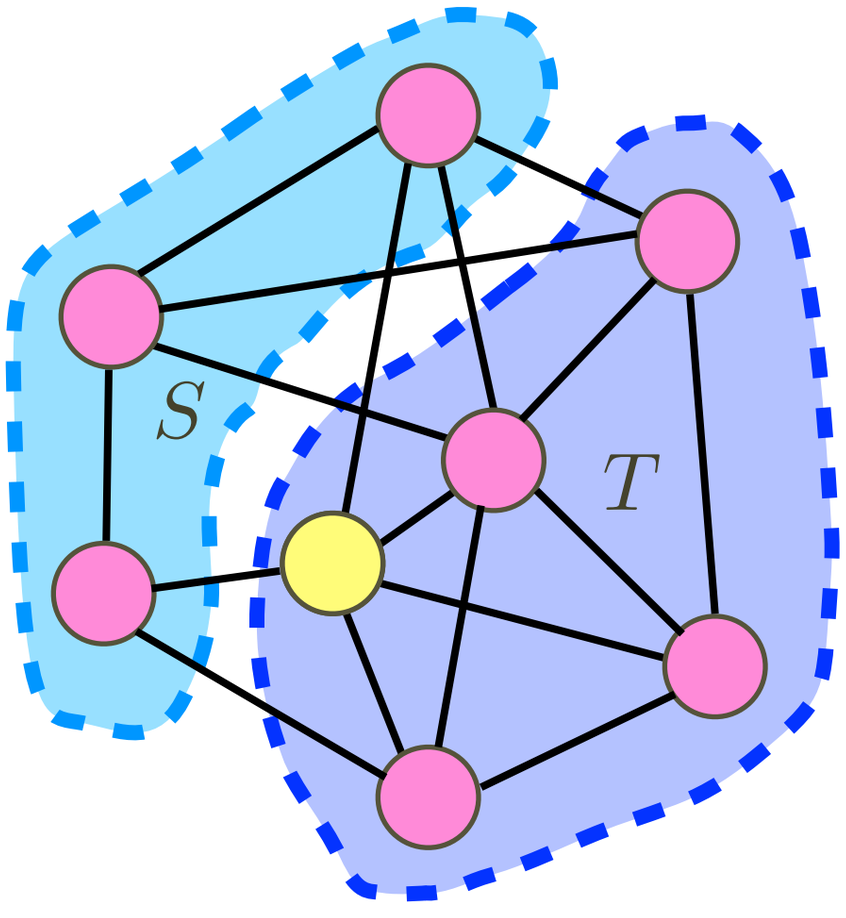
$V = \{v_1, v_2, \ldots, v_n\}$;
initially, $S=T=\emptyset$;
for $i = 1,2, \ldots, n$
  v_i joins one of S, T
  to maximize *current* $E(S,T)$

GreedyMaxCut is $1/2$-approximate

Max-Cut

Instance: An undirected graph $G(V, E)$.

Solution: A bipartition of V into S and T that maximizes the cut $E(S,T) = \{uv \in E: u \in S, v \in T\}$.

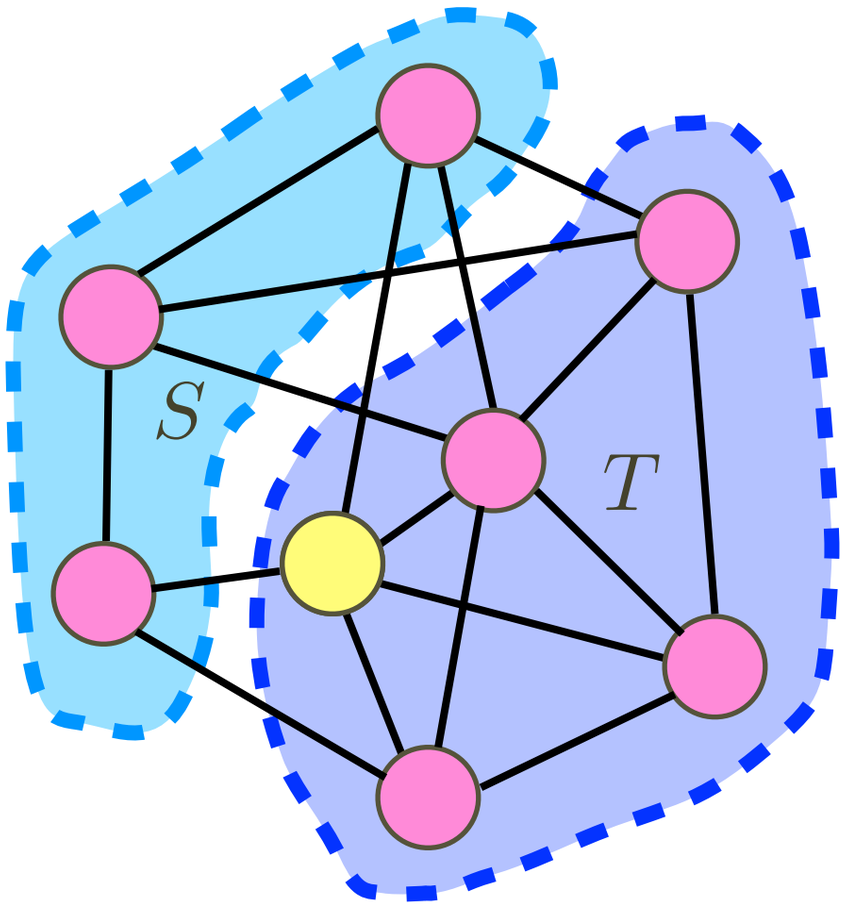


Local Search

Start with an *arbitrary* bipartition;
repeat until nothing changed:
  if $\exists v$ flipping side will increase cut
    v moves to the other side;

Local Search

Start with an *arbitrary* bipartition;
repeat until nothing changed:
  if $\exists v$ flipping side will increase cut
    v moves to the other side;



in a local optimum:

$$\forall\, v \in S,\ |E(v, S)| \le |E(v,T)|$$

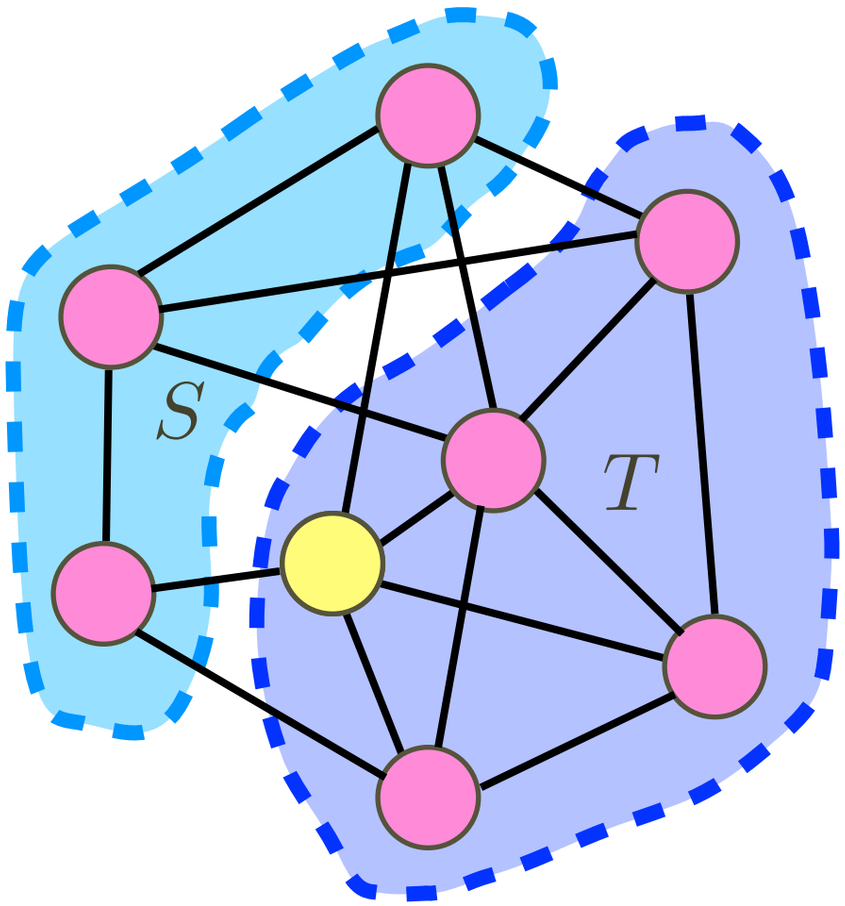
$$\Rightarrow\quad 2|E(S, S)| \le |E(S,T)|$$

$$\forall\, v \in T,\ |E(v, T)| \le |E(v,S)|$$

$$\Rightarrow\quad 2|E(T, T)| \le |E(S,T)|$$

Local Search

Start with an *arbitrary* bipartition;
repeat until nothing changed:
  if $\exists v$ flipping side will increase cut
    v moves to the other side;



in a local optimum:

$$\forall\, v \in S,\ |E(v, S)| \le |E(v,T)|$$

$$\Rightarrow\quad 2|E(S, S)| \le |E(S,T)|$$

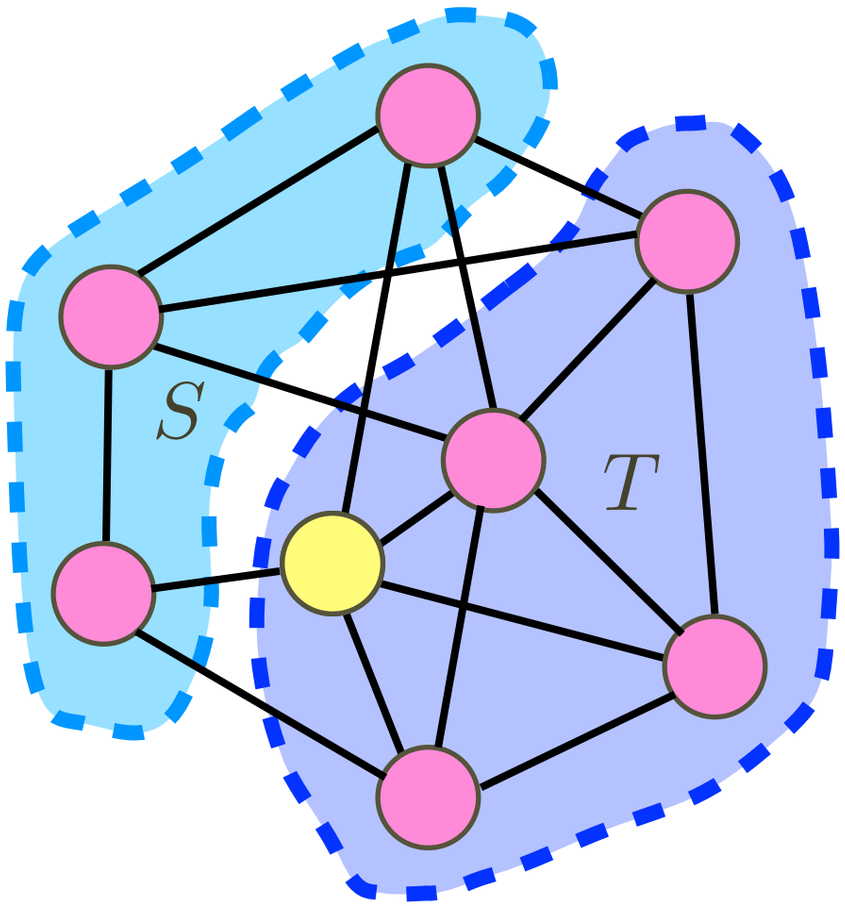
$$\forall\, v \in T,\ |E(v, T)| \le |E(v,S)|$$

$$\Rightarrow\quad 2|E(T, T)| \le |E(S,T)|$$

$$|E(S, S)| + |E(T, T)| \le |E(S,T)|$$

Local Search

Start with an *arbitrary* bipartition;
repeat until nothing changed:
  if $\exists v$ flipping side will increase cut
    v moves to the other side;



in a local optimum:

$$\forall\, v \in S,\ |E(v,S)| \le |E(v,T)|$$

$$\Rightarrow\quad 2|E(S,S)| \le |E(S,T)|$$

$$\forall\, v \in T,\ |E(v,T)| \le |E(v,S)|$$

$$\Rightarrow\quad 2|E(T,T)| \le |E(S,T)|$$

$$|E(S,S)| + |E(T,T)| \le |E(S,T)|$$

$$\text{OPT} \le |E| = |E(S,S)| + |E(T,T)| + |E(S,T)| \le 2|E(S,T)|$$

$$\Rightarrow\quad |E(S,T)| \ge \tfrac{1}{2} \cdot OPT$$

Local Search

Start with an *arbitrary* bipartition;
repeat until nothing changed:
    if $\exists v$ flipping side will increase cut
        v moves to the other side;

in a local optimum: $|E(S,T)| \geq \frac{1}{2} \cdot OPT$

GreedyMaxCut

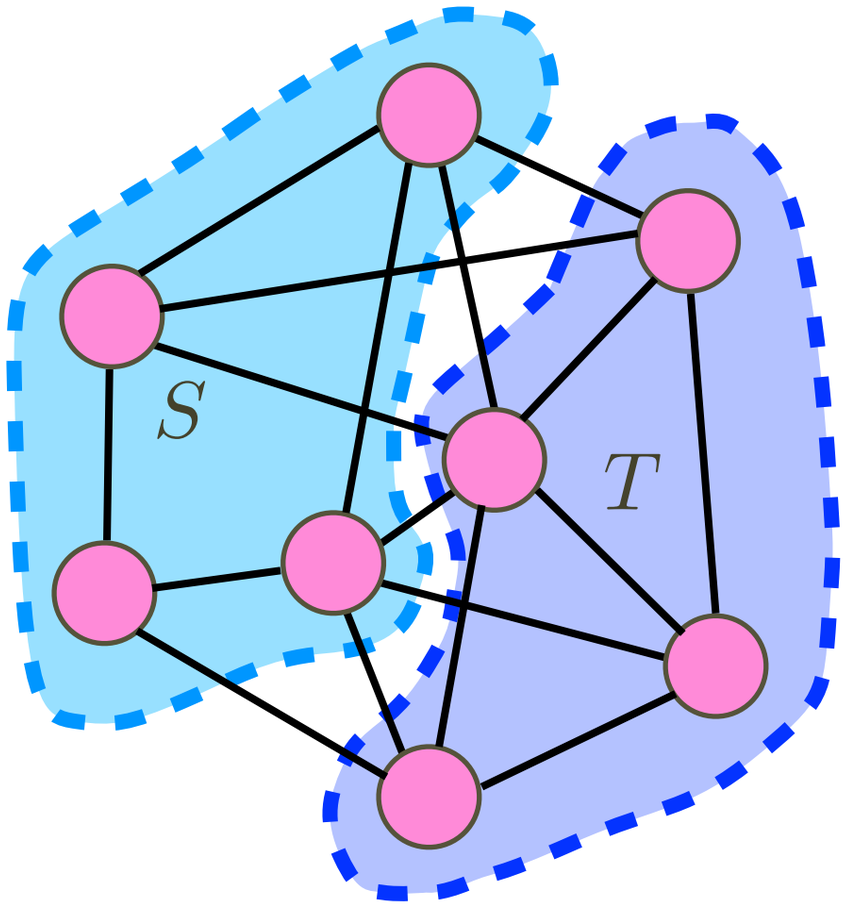
$V = \{v_1, v_2, \ldots, v_n\}$;
initially, $S=T=\emptyset$;
for $i = 1,2, \ldots, n$
    v_i joins one of S, T
    to maximize *current* $E(S,T)$

Is the cut returned by *GreedyMaxCut* locally optimal?

Max-Cut

Instance: An undirected graph $G(V, E)$.

Solution: A bipartition of V into S and T that maximizes the cut $E(S,T) = \{uv \in E: u \in S, v \in T\}$.



- **NP**-hard.
- *Greedy* and *local search* are 1/2-approximate.
- Rounding *semidefinite programing* has approximation ratio 0.878~.
- Assuming the *unique game conjecture*, no poly-time <0.878-approximate algorithm.

Maximum cut: big improvement flips

Local search. Within a factor of 2 for MAX-CUT, but not poly-time!

Big-improvement-flip algorithm. Only choose a node which, when flipped, increases the cut value by at least $\frac{2\varepsilon}{n} w(A, B)$

Claim. Upon termination, big-improvement-flip algorithm returns a cut (A, B) such that $(2 + \varepsilon) w(A, B) \geq w(A^*, B^*)$.

Pf idea. Add $\frac{2\varepsilon}{n} w(A, B)$ to each inequality in original proof. ▪

Claim. Big-improvement-flip algorithm terminates after $O(\varepsilon^{-1} n \log W)$ flips, where $W = \Sigma_e w_e$.

- Each flip improves cut value by at least a factor of $(1 + \varepsilon / n)$.
- After n / ε iterations the cut value improves by a factor of 2.
- Cut value can be doubled at most $\log_2 W$ times. ▪

if $x \geq 1$, $(1 + 1/x)^x \geq 2$

Neighbor relations for max cut

1-flip neighborhood. Cuts (A, B) and (A', B') differ in exactly one node.

k-flip neighborhood. Cuts (A, B) and (A', B') differ in at most k nodes.

KL-neighborhood. [Kernighan-Lin 1970]

- To form neighborhood of (A, B):
  - Iteration 1: flip node from (A, B) that results in best cut value (A_1, B_1), and mark that node.
  - Iteration i: flip node from (A_{i-1}, B_{i-1}) that results in best cut value (A_i, B_i) among all nodes not yet marked.
- Neighborhood of $(A, B) = (A_1, B_1), \ldots, (A_{n-1}, B_{n-1})$.
- Neighborhood includes some very long sequences of flips, but without the computational overhead of a k-flip neighborhood.
- Practice: powerful and useful framework.
- Theory: explain and understand its success in practice.

cut value of (A_1, B_1) may be worse than (A, B)

Outline: Local Search

- Gradient descent
- Metropolis algorithm and simulated annealing
- Hopfield neural networks
- Max-cut problem
- Take-home messages

Take-Home Messages

- Local search: modifying the state of the solution to its neighbor state, and check whether improvement appears.

- Usually finding local optimal solution instead of global optimum.

- It is tricky to define neighbor states and escape from local optimums.

- Further reading: the scheduling problem discussed in:

  - https://tcs.nju.edu.cn/wiki/index.php?title=%E9%AB%98%E7%BA%A7%E7%AE%97%E6%B3%95_(Fall_2020)/Greedy_and_Local_Search

  - https://tcs.nju.edu.cn/slides/aa2020/Greedy.pdf

Thanks for your attention!
Discussions?

Reference

Algorithm design: Chap. 12.

Slides from Kevin Wayne: https://www.cs.princeton.edu/~wayne/kleinberg-tardos/

Yitong Yin's lecture:

https://tcs.nju.edu.cn/wiki/index.php?title=%E9%AB%98%E7%BA%A7%E7%AE%97%E6%B3%95_(Fall_2020)/Greedy_and_Local_Search

https://tcs.nju.edu.cn/slides/aa2020/Greedy.pdf

Mackay book Chap. 42 on Hopfield nets: https://www.inference.org.uk/itprnn/book.pdf